AFRL-IF-RS-TR-2001-158
Final Technical Report
August 2001

TRUSTED RECOVERY FROM INFORMATION ATTACKS

George Mason University

Sushil Jajodia, Paul Ammann, and Peng Liu

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2001-158 has been reviewed and is approved for publication.

APPROVED: Joseph V Giordano

JOSEPH V. GIORDANO
Project Engineer

FOR THE DIRECTOR:

WARREN H. DEBANY, Technical Advisor
Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** AUGUST 2001	**3. REPORT TYPE AND DATES COVERED** Final Mar 97 - Mar 99

4. TITLE AND SUBTITLE
TRUSTED RECOVERY FROM INFORMATION ATTACKS

5. FUNDING NUMBERS
C - F30602-97-1-0139
PE - 61102F
PR - 2301
TA - 01
WU - 01

6. AUTHOR(S)
Sushil Jajodia, Paul Ammann, and Peng Liu

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
George Mason University
4400 University Drive
Fairfax VA 22030-4444

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory/IFGB
525 Brooks Road
Rome New York 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2001-158

11. SUPPLEMENTARY NOTES
Air Force Research Laboratory Project Engineer: Joseph V. Giordano/IFGB/(315) 330-4199

12a. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Preventive measures sometimes fail to deflect malicious attacks. In this work, we adopt an information warfare perspective which assumes success by the attacker in achieving partial, but not complete damage. In particular, we work in the database context and consider recovery form malicious but committed transactions. Traditional recovery mechanisms do not address this problem, except for complete rollbacks, which undo the work of benign transactions as well as malicious ones, and compensating transactions, whose utility depends on application semantics. recovery is complicated by the presence of benign transactions that depend, directly or indirectly, on the malicious transactions. We present recovery models to restore only the damaged part of the database. Two families of new repair algorithms are developed: one is a set of dependency-graph based algorithms, the other is a set of algorithms that do repair via rewriting histories.

14. SUBJECT TERMS
Defensive Information Warfare, Information Assurance

15. NUMBER OF PAGES
116

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	**18. SECURITY CLASSIFICATION OF THIS PAGE** UNCLASSIFIED	**19. SECURITY CLASSIFICATION OF ABSTRACT** UNCLASSIFIED	**20. LIMITATION OF ABSTRACT** UL

Contents

List of Figures

SECTION 1

Introduction

Information security has become increasingly important with the advent of (inter-connected) computers to process sensitive information. However, experience with traditional information systems security practices (INFOSEC) for confidentiality, integrity, and availability has shown that it is very difficult to adequately anticipate the abuse and misuse to which an information system will be subjected in the field. In 1995 the Computer Emergency Response Team (CERT) reported 1,168 security-related incidents [cer95]. That year the United States Federal Bureau of Investigation (FBI) disclosed the results of their computer security survey, which showed that 40 percent of the surveyed sites experienced at least one unauthorized access [fbi97]. In 1996 the United States Department of Defense (DoD) reported an estimate of 250,000 attacks per year on its computer system and stated that the rate of attacks is increasing by 100 percent annually [dod96]. That year CERT's figures showed a significant increase in hacker activity, with 2,573 security-related incidents [cer96].

In response to this experience, a complementary approach with an emphasis on survivability has emerged.* This 'information warfare' perspective is that not only should vigorous information security measures be taken to defend a system against attack, but that some attacks should be assumed to succeed, and that countermeasures to these successful attacks should be planned in advance. The information warfare perspective emphasizes the ability to live through and recover from attacks.

The focus of INFOSEC is prevention: security controls aim to prevent malicious activity that interferes with either confidentiality, integrity, or availability. However, outsiders (hackers) have proved many times that security controls can be breached in imaginative and unanticipated ways. Further, insiders have significant privileges by necessity, and so are in a position to inflict damage. The dramatic increase in internetworking has led to a corresponding increase in the opportunities

*For a summary with an emphasis on the database context, see [AJMB97].

for outsiders to masquerade as insiders. Network-based attacks on many systems can be carried out from anywhere in the world. Although mechanisms such as firewalls reduce the threat of outside attack, in practice such mechanisms do not eliminate the threat without blocking legitimate use as well. In brief, strong prevention is clearly necessary, but less and less sufficient, to protect information resources.

An information warfare approach augments traditional INFOSEC measures to harden a system against attack. An information warfare timeline is intelligence gathering by the adversary to detect weaknesses in the resulting system, attack by the adversary, and finally countermeasures to the attack. Typical countermeasure phases follow a fault tolerance model of attack detection, damage confinement and assessment, reconfiguration, damage repair, and fault treatment to prevent future similar attacks.

Although the information warfare adversary may find many weaknesses in the diverse components of an information system, databases provide a particularly inviting target. There are several reasons for this. First, databases are very widely used, so the scope for attack is large. Second, information in databases can often be changed in subtle ways that are beyond the detection capabilities of the typical database mechanisms such as range and integrity constraints. For example, repricing merchandise is an important and desirable management function, but it can easily be exploited for fraudulent purposes. Finally, unlike most system components, many databases are explicitly optimized to accommodate frequent updates. The interface provides the outside attacker with built in functions to implement an attack; all that is necessary is to acquire sufficient privileges, a goal experience has shown is readily achievable. Advanced authorization services can reduce such a threat, but never eliminate it, since insider attacks are always possible.

Integrity, availability, and (to a lesser degree) confidentiality have always been key database issues, and commercial databases include diverse set of mechanisms towards these ends. For example, access controls, integrity constraints, concurrency control, replication, active databases, and recovery mechanisms deal well with many kinds of mistakes and errors. However, the IW attacker can easily evade some of these mechanisms and exploit others to further the attack. For example, access controls can be subverted by the inside attacker or the outside attacker who has assumed an insider's identity. Integrity constraints are weak at prohibiting plausible but incorrect data; classic examples are changes to dollar amounts in billing records or salary figures. To a concurrency control mechanism, an attacker's transaction is indistinguishable from any other transaction. Automatic replication facilities and active database triggers can serve to spread the damage introduced by an attacker at one site to many sites. Recovery mechanisms ensure that committed transactions appear in stable storage and provide means of rolling back a database, but no attention is given to distinguishing legitimate activity from malicious activity. In brief, by themselves, existing database mechanisms for managing integrity, availability, and confidentiality are inadequate for detecting, confining, and recovering from IW attacks.

Massive IW attacks have large scale, immediate impact and consequently generate an immediate response. More insidious IW attacks inflict damage incrementally and open up the threat that the transactions of legitimate users can spread the damage throughout the database over an extended period of time before anyone notices that something is amiss. The longer the time period between attack and detection, the less satisfactory it is to roll back the database to a 'clean' state; too many transactions that performed useful, uncorrupted work are lost. There is a need to undo corrupted work without losing good work. Distinguishing transactions that read corrupted values from one that didn't isn't possible with current systems because 'read-from' dependency information is not maintained.

In some cases, the attacker's goal may be to reduce availability by attacking integrity. In the scenario outlined above, the attacker's goal not only introduces damage to certain data items and uncertainty about which good transactions can be trusted, but also achieves the goal of bringing the system down while repair efforts are being made. 'Coldstart' semantics for recovery mean that system activity is brought to a halt while damage is being repaired. To address the availability threat, recovery mechanisms with 'warmstart' or 'hotstart' semantics are needed. Warmstart semantics for recovery allow continuous, but degraded, use of the database while IW damage is being repaired. Hotstart semantics make recovery transparent to the users.

In this report, we focus on one specific countermeasure phase to an information warfare attack, namely the damage repair phase. We confine ourselves to the database context, and focus on mechanisms suitable for inclusion in commercial database systems.

1.1 Dissemination of Results from Contract

Before describing the results of the contract in detail, we describe the extent to which results from the contract have been disseminated in the literature. These results are grouped into the following areas:

1. Database Recovery Work: This area covers the main thrust of the contract, and focuses on specific mechanisms for restoring databases that have suffered damage from malicious information attacks.

2. Fault Tolerance Perspective: This area covers a higher level view of the entire survivability problem, as opposed to focusing on the recovery phase, as is done in the main body of the work.

3. Workshop Efforts: This area covers efforts by the authors to supply their work as input for a research agenda in the survivability area.

Database Recovery Work

This work comprises the main body of this report. Two in depth journal articles describe this work. One has been published; the other is still in review:

1. Peng Liu, Paul Ammann, and Sushil Jajodia. Rewriting histories: Recovering from malicious transactions. *The International Journal of Distributed and Parallel Databases.* 8(1):7-40, January 2000.

   Abstract: We consider recovery from malicious but committed transactions. Traditional recovery mechanisms do not address this problem, except for complete rollbacks, which undo the work of good transactions as well as malicious ones, and compensating transactions, whose utility depends on application semantics. We develop an algorithm that rewrites execution histories for the purpose of backing out malicious transactions. Good transactions that are affected, directly or indirectly, by malicious transactions complicate the process of backing out undesirable transactions. We show that the prefix of a rewritten history produced by the algorithm serializes exactly the set of unaffected good transactions. The suffix of the rewritten history includes special state information to describe affected good transactions as well as malicious transactions. We describe techniques that can extract additional good transactions from this latter part of a rewritten history. The latter processing saves more good transactions than is possible with a dependency-graph based approach to recovery.

2. Paul Ammann, Sushil Jajodia, and Peng Liu. Recovery from Malicious Transactions Under review with *IEEE Transactions on Knowledge and Data Engineering.*

   Abstract: Preventive measures sometimes fail to deflect malicious attacks. In this paper, we adopt an information warfare perspective, which assumes success by the attacker in achieving partial, but not complete, damage. In particular, we work in the database context and consider recovery from malicious but committed transactions. Traditional recovery mechanisms do not address this problem, except for complete rollbacks, which undo the work of benign transactions as well as malicious ones, and compensating transactions, whose utility depends on application semantics. Recovery is complicated by the presence of benign transactions that depend, directly or indirectly, on the malicious transactions. We present algorithms to restore only the damaged part of the database. We identify the information that needs to be maintained for such algorithms. The initial algorithms repair damage to quiescent databases; subsequent algorithms increase availability by allowing new transactions to execute concurrently with the repair process. Also, via a study of benchmarks, we show practical examples of how offline analysis can efficiently provide the necessary data to repair the damage of malicious transactions.

Fault Tolerance Perspective

1. Sushil Jajodia, Peng Liu, and Paul Ammann. A Fault Tolerance Approach to Survivability. In *Proceedings of the Information Systems Technology Symposium: Protecting NATO Information Systems in the 21st Century,* RTO/NATO, Hull, Canada (limited release), pages 20-1 to 20-7, October, 1999.

   Abstract: Attacks on computer systems have received a great deal of press attention; however, most of the focus has been on how an attacker can disrupt an organization's operations. Although attack prevention is clearly preferred, preventive measures do fail, and some attacks inevitably succeed in compromising some or all of particular systems. We adopt a fault tolerance approach that addresses all phases of survivability: attack detection, damage confinement, damage assessment and repair, and attack avoidance, but we give special attention to recovery issues, and how recovery from malicious activity can be planned for and executed. For specific examples, we discuss recovery models for backing out malicious, committed transactions, either syntactically using read-from dependencies, or via rewriting histories, which can save the work of additional good transactions.

2. Sushil Jajodia, Catherine D. McCollum, and Paul Ammann. Trusted recovery: An important phase of information warfare defense. *Communications of the ACM.* 42(7):71-75, July 1999.

   Abstract: Information warfare defense involves not just protective mechanisms but also detection and reaction to successful attacks and a process for managing the tracking, containment, and recovery from damage. Unlike many hackers, who may wish to provide evidence of their entry into a system, information warfare attackers may pursue a more subtle course directed towards serious harm to an organization's ability to meet its mission rather than an obvious, temporary disruption. Such an attack could target not just the system or network itself, but also the information upon which an organization relies. Information warfare attacks can spoof legitimate users or make use of malicious insiders, so information warfare defense must also incorporate techniques effective against insider attack. This paper describes the cycle of activity involved in information warfare defense. It then discusses a framework for detecting, managing, and recovering from damage inflicted by information warfare on the critical information maintained within the system. Allowing system operation to proceed while some information is damaged and under repair has implications for maintaining consistency, so a modified consistency model is presented. Finally, a variety of methods for recovery and containment are discussed that can be used depending on the characteristics of the system and severity of the damage.

3. Sushil Jajodia, Paul Ammann, and Catherine D. McCollum. Surviving information warfare attacks. *IEEE Computer.* 32(4):57-63, April 1999.

   Abstract: Information warfare has received a great deal of attention in the press lately; however, most of the focus has been on how an attacker can disrupt an organization's operations. In this paper, we discuss issues and methods for survivability of systems under malicious attacks, with particular emphasis on the information elements of systems. Although attack prevention is clearly preferred, preventive measures do fail, and some attacks inevitably succeed in compromising some or all of particular systems. We discuss a fault-tolerance approach that can address all phases of survivability: attack detection, damage confinement, damage assessment and repair, and attack avoidance. We also discuss some mechanisms that can give systems the ability to live through and recover from successful attacks. Defensive information warfare is far from being a solved problem from the research perspective, let alone the practical perspective. Consequently, a major goal of this paper is to raise awareness among system developers of the need to include information warfare considerations in system analysis and design.

4. Paul Ammann, Sushil Jajodia, Catherine D. McCollum, and Barbara T. Blaustein. Surviving information warfare attacks on databases. In *Proceedings 1997 IEEE Computer Society Symposium on Security and Privacy,* pages 164-174, Oakland, CA, May 1997.

   Abstract: We consider the problem of surviving information warfare attacks on databases. We adopt a fault tolerance approach to the different phases of an information attack. To maintain precise information about the attack, we mark data to reflect the severity of detected damage as well as the degree to which the damaged data has been repaired. To increase availability we introduce a marking for partially repaired data. In this case, integrity constraints might be violated, but the data is nonetheless usable to support mission objectives. We define a notion of consistency suitable for databases in which some information is known to be damaged, and other information is known to be only partially repaired. We present a protocol for normal transactions with respect to the damage markings and show that correct normal transactions that follow the protocol maintain database consistency. We present an algorithm for taking consistent snapshots of databases under attack. The snapshot algorithm has the virtue of not interfering with countermeasure transactions.

Workshop Efforts

1. Paul Ammann, Bruce H. Barnes, Sushil Jajodia, and Edgar H. Sibley, editors. *Proceedings of CSDA 98: Computer Security, Dependability, and Assurance: From Needs to Solutions,* IEEE Press, Los Alamitos, CA, pages 204-212, 1999.

Summary: This ONR/NSF funded workshop brought together leading researchers to investigate the intersection of three related areas: Computer Security, Dependability, and Assurance. The PIs for this report edited the proceedings.

2. Paul Ammann and Sushil Jajodia. Computer Security, Fault Tolerance, and Software Assurance. *IEEE Concurrency,* 7(1):4-6, January-March 1999.

   Summary: This is a widely disseminated description of the CSDA 98 Workshop listed in the prior item in this section.

3. Paul Ammann, Sushil Jajodia, and Peng Liu. A Fault Tolerance Approach to Survivability. In *Proceedings of CSDA 98: Computer Security, Dependability, and Assurance: From Needs to Solutions,* IEEE Press, Los Alamitos, CA, pages 204-212, 1999. ISBN 0-7695-0337-3.

   Abstract: (Note: This is a contribution to the workshop listed in the first item of this section.) Attacks on computer systems have received a great deal of press attention; however, most of the focus has been on how an attacker can disrupt an organization's operations. Although attack prevention is clearly preferred, preventive measures do fail, and some attacks inevitably succeed in compromising some or all of particular systems. We propose research into a fault-tolerance approach that addresses all phases of survivability: attack detection, damage confinement, damage assessment and repair, and attack avoidance. We focus attention on continued service and recovery issues. A promising area of research for continued service addresses relaxed notions of consistency. Expanding on the notion of self stabilization, the idea is to formalize the degree of damage under which useful services is still possible. A complementary research area for recovery is the engineering of suitable mechanisms into existing systems. We explain the underlying models for these research areas and illustrate them with examples from the database domain. We argue that these models form a natural part of a fault tolerance approach and propose research into adapting these models for larger systems.

1.2 Research Results

The report describes the following results. First, this report describes two novel recovery models to bridge the theoretical gap between classical database recovery theory where only uncommitted transactions can be undone, and trusted recovery practice where operations with the same (operational) semantics as traditional undos are needed to remove the effects of such committed transactions as malicious transactions and affected benign transactions (For simplicity, we use the same word, namely 'undo', to denote such operations). In particular, this report describes

(1) a *flat-transaction* recovery model where committed transactions are 'undone' by building and executing a specific type of transactions, namely, *undo* transactions, and (2) a *nested-transaction* model where a flat commercial history is virtually extended to a two-layer nested structure where originally committed transactions turn out to be subtransactions hence traditional undo operations can be directly applied to the model without violating the durability property.

Second, this report provides a family of syntactic recovery algorithms that, given a specification of malicious, committed transactions, unwinds the effects of each malicious transaction, along with the effects of any benign transaction that depends, directly or indirectly on a malicious transaction. Significantly, the work of the remaining benign transactions is saved. The first algorithm yields coldstart semantics; the database is unavailable during repair. The second algorithm yields warmstart semantics; normal use may continue during repair, although some degradation of service may be experienced by some transactions. Moreover, this report outlines various possibilities for maintaining read-from dependency information. Although direct logging of transaction reads has the virtue of simplicity, the performance degradation of such an approach may be too severe in some cases. For this reason, this report shows that offline analysis can efficiently meet the need for establishing read-from dependency information. This report illustrates the practicality of such an approach via a study on standard benchmarks.

Third, this report presents an algorithm that rewrites an execution history for the purpose of backing out malicious transactions. Good transactions that are affected, directly or indirectly, by malicious transactions complicate the process of backing out undesirable transactions. This report shows that the prefix of a rewritten history produced by the algorithm serializes exactly the set of unaffected good transactions, thus is equivalent to using a write-read dependency graph approach. The suffix of the rewritten history includes special state information to describe affected good transactions as well as malicious transactions. This report describes techniques that can extract additional good transactions from the latter part of a rewritten history. The latter processing saves more good transactions than is possible with a dependency-graph based approach or a commutativity based approach to recovery.

Although we develop the above algorithms to repair a database when some malicious activity happens, our methods can be easily extended to other applications where some committed transactions may also be identified undesirable, thus have to be backed out. For example

- In [JLM98], the use of isolation is proposed to protect systems from the damage caused by authorized but malicious users, masqueraders, and misfeasors, where the capacity of intrusion detection techniques is limited. In the database context, the basic idea is when a user is found suspicious, his transactions are redirected to an isolated database version, and if the user turns out to be innocent later, the isolated database version will be merged into the main database version. Since these two versions may be inconsistent, some committed transactions may have to be backed out to ensure the consistency of the database.

- During upgrades to existing systems, particularly upgrades to software. Despite efforts for planning and testing of upgrades, upgrade disasters occur with distressing regularity.[†] If a system communicates with the outside world, bringing the upgrade online with a hot standby running the old software isn't complete protection. Problems with an upgrade by one organization can easily affect separate, but cooperating organizations. Thus an incorrect upgrade at a given organization may result in an erroneous set of transactions at one or more cooperating organizations. In many cases, it is not possible simply to defer activity, and so during the period between the introduction of an upgrade and the recognition of an upgrade problem, erroneous transactions at these cooperating organizations commit. As a result, backing out these committed erroneous transactions is necessary.

- In partitioned distributed database systems, Davidson's optimistic protocol [Dav84] allows transactions to be executed within each partitioned group independently with communication failures existing between partitioned groups. As a result, serial history H_i consisting of all transactions executing within group P_i is generated. When two partitioned groups P_1 and P_2 are reconnected, H_1 and H_2 may conflict with each other. Therefore, some committed transactions may have to be backed out to resolve the conflicts and ensure the consistency of the database.

- In [GHOS96], J. Gray et al. state that update anywhere-anytime-anyway transactional replication has unstable behavior as the workload scales up. To reduce this problem, a two-tier replication algorithm is proposed that allows mobile applications to propose tentative update transactions that are later applied to a master copy. The drawback of the protocol is that every tentative transaction must be reexecuted on the base node, thus some sensitive transactions may have given users inaccurate information and the work of tentative transactions is lost. In this situation, the strategy that when a mobile node is connected to the base node merges the mobile copy into the master copy may be better, however, in order to ensure the consistency of the master copy after the mergence, some committed transactions may have to be backed out.

[†] For some more spectacular examples, see Peter Neumann's RISKS digest in the newsgroup `news:comp.risks` or the archive `ftp://ftp.sri.com/risks`.

1.3 Organization of the Report

The outline of the report is as follows. Chapter 2 describes related work. In Chapter 3, we present a model for a database system that can survive information warfare attacks, including its transaction processing features. Chapter 4 presents two recovery models to support 'undoing' undesirable committed transactions, such as malicious transactions and affected good transactions. In Chapter 5, we present a syntactic repair model where both coldstart and warmstart recovery algorithms are developed. Moreover, we use benchmark applications to show how offline analysis can mitigate performance degradation during normal operation. In Chapter 6, we present a repair model based on history rewriting, where we first give a rewriting algorithm and show that it is equivalent to using a dependency-graph based approach; we second turn to methods to save additional good transactions; we third show how to prune a rewritten history so that a repaired history can be generated; moreover, we examine the relationships among the possible rewriting algorithms; finally, we show how to implement the rewriting model in a realistic transaction processing system which is based on the Saga model [GMS87]. In Chapter 7, we discuss some issues relevant to our repair model, enumerate the contributions of this dissertation and present an insight into future research directions.

SECTION 2

Related Work

Recovery methods have been studied extensively by researchers in fault tolerance and in database areas. After a comprehensive introduction of the limitations of traditional mechanisms in doing trusted recovery, this chapter first addresses the related work in the area of fault tolerance, then addresses the related work in the area of databases. Some related work in the areas of computer security and information warfare is also addressed.

2.1 Why Traditional Mechanisms Fail in Trusted Database Recovery

Although recovery methods have been studied extensively by researchers in fault tolerance (e.g., see [LA90, RLKL95]) and in database areas (e.g., see [Dat95, Dat83, GR93, RC97, MHL$^+$92, HR98]), the existing methods work well in case of failures under normal conditions. Achieving recovery under an information attack is clearly more difficult since the attack is malicious in nature and the attacker can be assumed to be familiar with the intricacies of the system being attacked. Therefore, achieving recovery requires modifications and extensions of existing techniques together with novel techniques that are only suitable for surviving information attacks.

In fault tolerance area [LA90, RLKL95], two types of errors are considered: errors that are *anticipated* and those that are *unanticipated.* In the case of anticipated errors, an accurate prediction or assessment of the damages can be made; if this is not possible, errors are said to be unanticipated.

An example of an anticipated error is the loss or duplication of a message, perhaps due to an unreliable communication link, or perhaps due to a malicious attacker who has intercepted the link. Anticipating link failures can be accomplished by providing redundant links. Anticipating link intercepts can be accomplished by providing special information in the message being sent.

In the case of a link failure, if careful attention is paid to joint failure modes such as a common intermediate node in a network, it is possible to reliably recover from the lost message by resending the message over the redundant channel.

A different example of an error that can be anticipated is a value out of range during a type conversion, for example, from floating point to integer. Recovery can be achieved through the prudent use of exception handlers. Failure to do so can be costly, as demonstrated by the ill-fated maiden flight of the Ariane 5, which was lost shortly after take-off due to events that were traced back to a type conversion that was not protected by an exception handler.

To recover from anticipated errors, *forward recovery* methods are used. Since the errors have been foreseen, either contingency update instructions can be specified or a means of deriving an acceptably correct value can be formulated. Both examples mentioned above, link failures and type conversion errors, are well suited to forward recovery methods.

Forward recovery methods have two limitations. First, these methods are usually very system specific. Second, success of these methods depends on how accurately damages from faults can be predicted and assessed. Therefore, current forward recovery mechanisms can not be directly applied to a specific database system where information attacks are usually difficult to be predicted or assessed.

To recover from unanticipated errors, *backward recovery* is considered to be the only viable approach. This requires that the entire state be replaced by a prior state that is consistent. Clearly, this approach is less than optimal because it requires that the system be halted temporarily. As observed earlier, this in itself may be the attacker's objective, particularly if the attacker can cause it to occur at a critical time.

Database management systems (DBMSs) provide a rich set of recovery facilities [Dat95, Dat83, GR93, HR83, RC97, MHL$^+$92]. These facilities require a clear understanding of the following two factors:

- What are the correct database states since they determine when recovery is necessary
- What kinds of failures are expected and their characteristics

Whether a database state is correct or not is determined as follows: A database has associated with it a collection of integrity constraints. A database state is said to be correct if it satisfies the associated integrity constraints. DBMSs provide some support for specifying integrity constraints. Examples are primary key constraints, referential integrity constraints, and range constraints.

Kinds of failures that are considered fall into these broad categories:

- *Transaction Failures:* A transaction may abort because it is requested by the user or because it is forced by the system. The later may be the case if the transaction violates some consistency constraint or is involved in a deadlock.

- *System Failures:* These are failures that are caused by a fault in the software.
- *Storage Media Failures:* These failures include volatile storage (main memory and paging space), non-volatile on-line storage (database and log disks), and non-volatile off-line storage (e.g., tapes).
- *Communication Failures:* These are failures in communication between two nodes of a distributed system.

To combat errors in the database, any transaction that violates the integrity constraints is aborted, in which case the database state stays correct and there is no need for further recovery. All other failures are considered unanticipated, and database recovery facilities mostly rely on backward recovery methods to restore the database to a consistent state. Although forward recovery by executing compensating transactions [GMS87, AJR97] is possible, this is considered highly application dependent and, therefore, is not provided any support by the system.

Backward recovery in databases is performed by implementing two basic operations - *undo* and *redo* - on the *stable* database (i.e., the state of the database on non-volatile storage). An undo operation undoes updates by an aborted transaction to the stable database, while a redo operation redoes the updates by a committed transaction to the stable database.

Although all these features deal well with many kinds of errors and system failures, their effectiveness against an information warfare attacker is limited [AJMB97]. Information warfare defense must consider the possibility that authorization controls could be defeated; that an authorized user, through greed, disgruntlement, or ideology, might become an attacker; or that an attacker might gain the use of a legitimate user's identity, with the corresponding authorizations. Any of these scenarios might result in the intentional corruption of the database by the introduction of incorrect or misleading data. Then, not only are some of these controls ineffectual against the problem, but those intended to maintain consistency among related data may help to spread the contamination.

For example, entity and range constraints can ensure that individual data values exist and are legal, but they cannot guarantee that these values are reasonable or accurate for the particular entity being described. An attacker could disrupt functions that depend on the database either by inserting a wrong value for particularly critical data or by distorting the overall picture to render aggregates or frequency distributions significantly inaccurate by small changes to many individual items. Referential constraints ensure that interrelationships among entities are maintained, but an attacker could easily make corresponding changes in related data entities. If cascade or delete rules have been specified for the referential integrity constraints, they may actually assist the attacker, spreading the problem by making the corresponding changes automatically. Concurrency controls ensure only that malicious transactions are properly scheduled along with others. Automated replication helps keep data available in a distributed system in the face of individual system failures, but also serves as an efficient means of spreading erroneous data.

There are several limitations to the backward recovery methods used in DBMSs, especially in face of malicious attacks. First, if a transaction is aborted, the transaction isolation property supports recovery, in a sense, by ensuring that it can be backed out without affecting other transactions. This would not arise, however, in the case of a malicious transaction, because it would appear to the DBMS like any other transaction and would complete normally. Undo/redo logs support recovery when the system fails with a number of uncompleted transactions in progress, but this also does not arise when transactions complete successfully but create bad data. Now, suppose that at some time after a malicious transaction has completed and been committed, the bad data it created is discovered through some means. (Perhaps a human user has noticed it.) Meanwhile, other innocent transactions may have read the bad data, based their computations on it, and unwittingly then written bad data of their own to other items (Informally, we say these innocent transactions are *affected*). The only general mechanism available to remove the effects of one or more prior, successfully committed transactions is backward recovery, which rolls the database back to a previously established checkpoint. However, the use of this mechanism poses a dilemma, because the penalty for doing so is that all other, valid work that has been accomplished since the checkpoint was taken is also lost.

2.2 Related Work in Fault Tolerance

Recovery in fault tolerance focuses on *error recovery* with the purpose of eliminating errors from the system state [LA90, RLKL95]. Error recovery techniques can be classified into two categories: *backward error recovery* techniques and *forward error recovery* techniques. Backward error recovery techniques restore a prior state of a system in the hope that the earlier state will be error free. In contrast, forward error recovery techniques manipulate some portion of the current state to produce a new state, again in the hope that the new state will be error free. As we mentioned in Chapter 1, backward recovery methods can cause too much rework, that is, the work of many good transactions may be lost; and forward recovery methods are usually very system specific, and the success of these methods depends on how accurately damages from faults can be predicted and assessed.

Although execution of malicious transactions may not generate *errors* (malicious transactions can easily transform consistent states to consistent states), error recovery techniques can be adapted to do attack recovery by viewing a malicious transaction as a component with a *fault*, thus the state transition produced by the transaction can be viewed as the *manifestation* of the fault, and the updates of the transaction can be viewed as errors produced by the manifestation. Besides database recovery mechanisms which we will address in next section, specific error recovery methods have been proposed in many scenarios such as electronic switching systems (ESS) [KQ72], critical computer systems [KQ72, ALS78], program executing [TB82, Ber88], and cooperating processes [Ran77].

Error correcting codes [PW72] are widely used in computer systems to provide recovery from anticipated faults affecting memory units. Error correcting codes use redundancy to enable the position of the erroneous bit(s) to be calculated, its value re-inverted and thereby avert a failure of the memory. However, error correcting codes are not useful to attack recovery because state transitions produced by malicious transactions are often valid.

In [TB82], a theory for the use of *structural redundancy* in data structures as a means of recovering from structural damage is developed. The redundant information can be checked for consistency, and this structure is corrected if inconsistent. However, redundant storage structures can not be used to detect and recover from damages caused by malicious transactions because execution of malicious transactions does not make the database state inconsistent.

In [Ber88], *recovery points* are automatically established by a processor to provide tolerance of CPU failures. A recovery point is a point in time during the activity of a system for which the then current state may subsequently need to be restored. A recovery point is *established* by arranging that appropriate information is preserved so that at any subsequent time it will be possible to restore the recovery point. The idea of restoring recovery points is similar to that of checkpointing. However, restoring the database state to its latest checkpoint may unnecessarily lose the work of many good transactions.

In [Ran77], recovery for cooperating processes is studies and it is found that the attempts to achieve backward error recovery can result in the *domino effect* problem. The domino effect of cascading rollback can seriously damage the system performance. Although *synchronous* checkpointing can avoid the domino effect, it is undesirable in many situations. To ensure progress in *asynchronous* checkpointing, *message logging* is adopted in various recovery protocols [BBG83, SY85, JZ90]. In [LA94], message semantics is exploited to reduce rollback in optimistic message logging recovery schemes. In particular, semantic relationships between operations indicated by messages are used to identify insignificant messages which can be logically removed from the computation without changing its meaning or result. Viewing transactions as processes, this report is similar to [LA94] in the sense that they both aim to reduce rollback overhead by exploiting the dependencies between processes (transactions). However, they are significantly different: (1) they address problems in different contexts, thus their models are very different; (2) they exploit different kinds of syntactic dependencies; (3) although commutativity is also exploited in [LA94], this report extends commutativity to a new kind of dependencies, denoted *can precede*, which is not addressed in [LA94]; (4) the rewriting techniques proposed in this report are not addressed in [LA94].

2.3 Related Work in Databases

Database recovery is one of the best success stories of software fault tolerance. However, database recovery mechanisms are not designed to deal with malicious attacks. Traditional recovery mechanisms [BHG87] based on physical or logical logs guarantee the ACID properties of transactions - Atomicity, Consistency, Isolation, and Durability - in the face of process, transaction, system and media failures. In particular, the last of these properties ensure that traditional recovery mechanisms never undo committed transactions. However, the fact that a transaction commits does not guarantee that its effects are desirable. Specifically, a committed transaction may reflect inappropriate and/or malicious activity.

Although our repair model is related to the notion of *cascading abort* [BHG87], cascading aborts only capture the *read-from* relation between active transactions. However, it may be necessary to capture the read-from relation between two committed transactions, even if the second transaction began long after the first one committed. In addition, in standard recovery approaches cascading aborts are avoided by requiring transactions to read only committed data [KLS90].

There are two common approaches to handling the problem of undoing committed transactions: rollback and compensation. The rollback approach is simply to roll back all activity - desirable as well as undesirable - to a point believed to be free of damage. Such an approach may be used to recover from inadvertent as well as malicious damage. For example, users typically restore files with backup copies in the event of either a disk crash or a virus attack. In the database context, checkpoints serve a similar function of providing stable, consistent snapshots of the database. The rollback approach is effective, but expensive, in that all of the desirable work between the time of the backup and the time of recovery is lost. Keeping this window of vulnerability acceptably low incurs a substantial cost in maintaining frequent backups or checkpoints, although there are algorithms for efficiently establishing snapshots on-the-fly [AJM95, MPL92, Pu86].

The compensation approach [GM83, GMS87] seeks to undo either committed transactions or committed steps in long-duration or nested transactions [KLS90] without necessarily restoring the data state to appear as if the malicious transactions or steps had never executed. There are two kinds of compensation: action-oriented and effect-oriented [KLS90, Lom92, WHBM90, WS92]. Action-oriented compensation for a transaction or step T_i compensates only the actions of T_i. Effect-oriented compensation for a transaction or step T_i compensates not only the actions of T_i, but also the actions that are affected by T_i. For example, consider a database system that deals with transactions that represent purchasing of goods. The effects of a purchasing transaction T_1 might have triggered a dependent transaction T_2 that issued an order to the supplier in an attempt to replenish the inventory of the sold goods. In this situation, the action-oriented compensating transaction for T_1 will just cancel the purchasing; but the effect-oriented compensating transaction for T_1 will cancel the order from the supplier as well. Although a variety of types of compensation are possible, all of them require semantic knowledge of the application.

The notion of commutativity, either of operations [LMWF94, Wei88, Kor83] or of transactions [SKPO88], has been well exploited to enhance concurrency in semantics-driven concurrency control. There are several types of commutativity. In operation level, for example, two operations O_1 and O_2 *commute forward* [Wei88] if for any state s in which O_1 and O_2 are both defined, $O_2(O_1(s)) = O_1(O_2(s))$; O_2 *commutes backward through* [LMWF94] O_1 if for any state s in which O_1O_2 is defined, $O_2(O_1(s)) = O_1(O_2(s))$; O_1 and O_2 *commute backward* [LMWF94, Wei88] if each commutes backward through the other. In transaction level, for example, two transactions *commute* [SKPO88] if any interleaving of the actions of the two transactions for which both transaction commit yields the same final state; Two transactions *failure commute*[SKPO88] if they commute, and if they can both succeed then a unilateral abort by either transaction cannot cause the other to abort. Our notation *can precede* is adapted from the *commutes backward through* notation for the purpose of taking advantage of transaction level commutativity.

In [BK92], semantics of operations on abstract data types are used to define *recoverability*, which is a weaker notion than commutativity. *recoverability* is a more general notion than *can follow* in capturing the semantics between two operations or transactions, but *can follow* is more suitable for rewriting histories. *recoverability* is applied to operations on abstract data types but *can follow* is applied to transactions. *recoverability* is defined based on the return value of operations, and thus a purely semantic notion; but *can follow* is defined based on the intersections of read and write sets of two transactions.

Korth, Levy, and Silberschatz [KLS90] address the recovery from undesirable but committed transaction. The authors build a formal specification model for compensating transactions which they show can be effectively used for recovery. In their model, a variety of types of correct compensation can be defined. A compensating transaction, whose type ranging from traditional undo, at one extreme, to application-dependent, special-purpose compensating transactions, at the other extreme, is specified by some constraints which every compensating transaction must adhere. Different types of compensation are identified by the notion of compensation soundness. A history X consisting of T, the compensating-for transaction; CT, the compensating transaction; and $dep(T)$, a set of transactions dependent upon T, is *sound* if it is equivalent to some history of only the transactions in $dep(T)$.

Though a compensating transaction in our model can be specified by their model, our notion of a *repaired* history is more suitable for rewriting histories than the notion of *sound* history, since the constraint that compensating transactions can only be applied to the final state of a history greatly decreases the possibility of finding a sound history, even if commutativity is fully exploited. We can get a feasible history by rewriting the original history based on *can follow*, *can precede*, *invert* and *cover*. The resulting history augmented with the corresponding undo-repair actions or fixed compensating transactions yields the desired repair.

2.4 Related Work in Security

Information in computer systems is vulnerable to several kinds of *threats*, namely actions or events that might prejudice security [Den83]. For example, threats to confidentiality include browsing, leakage, and inference; threats to integrity and availability include tampering and accidental destruction. The vulnerability not only incurs *information attacks* (*attacks* for brevity) which are the acts of trying to exploit it to degrade the security of computer systems, but also results in the development of *countermeasures* which are actions, devices, procedures, techniques, or other measures that reduce the vulnerability.

Discretionary access control (DAC), for example, is a widely used countermeasure, in which the owner of information determines at his or her discretion who else to share the information with. However, it is susceptible to to Trojan Horse attacks. A Trojan Horse is a malicious piece of code which is embedded within a host program. The Trojan Horse allows the host program to do its own job and has no visible effect on the latter's output. At the same time, however, the Trojan Horse does something malicious without directly violating the security rules of the system. The reason Trojan Horses work is because a program run by a user usually inherits the same unique ID, privileges and access rights as the user.

To conquer the vulnerability of DACs to Trojan Horse attacks, *mandatory access control* (MAC) was proposed by Bell and LaPadula in [BL76]. The Bell-LaPadula model divides the entities in a computer system into abstract sets of *subjects* and *objects*. An *object*, i.e., a record, a page, a file, etc., is a passive entity that contains or receives information. Access to an object potentially implies access to the information it contains. A *subject*, on the other hand, is an active entity, generally in the forms of a process of device that causes information to flow among objects. In addition, each object (subject) is associated with a mandatory *security class*, which can not be modified by any user process. A security class consists of two components - a hierarchical component called the *security level*, and a non-hierarchical component called the *category*. A *multilevel secure* (MLS) system is one which partitions its objects and subjects into security classes.

The Bell-LaPadula security policy can be summarized by the following two rules:

1. **Simple security property:** No subject may read information classified above its security level.

2. $\star$ **- property:** No subject may write information classified below its security level.

Although the MAC rules can prevent direct Trojan Horse attacks, information can still be leaked through what are known as *covert channels*. A covert channel is a communication channel based on usage of system resources that allows two cooperating processes to transfer information in a manner violating the security policy of the system. Two types of covert channels have been identified just far. They are *covert storage channels* and *covert timing channels.* Note that a covert channel is usually the result of a specific implementation of an algorithm (a protocol) rather than inherently present in the algorithm (protocol). However, sometimes such a communication channel is inherent to an algorithm (a protocol) and consequently appears in every implementation of the algorithm (protocol). This kind of communication channels are often denoted as *signaling channels.*

It has been found that signaling channels exist in classical transaction processing protocols (described in Chapter 3), especially concurrency control protocols, when a database system using these protocols is extended to a multilevel secure database system [AJB97]. Eliminating such signaling channels is one of the main challenges in developing a multilevel secure database system. Readers can refer to Section 7.1.1 for more relevant issues in multilevel secure transaction processing.

2.5 Related Work in Information Warfare

Although the area of IW defense is new, there is some relevant work. Graubert, Schlipper, and McCollum identified database management aspects that determine the vulnerability to information warfare attacks [GSM96]. McDermott and Goldschlag [MG96a, MG96b] developed storage jamming, which can be used to seed a database with dummy values, access to which indicates the presence of an intruder. Although data jamming is primarily intended for detection, it could also help deceive the attacker and confuse the issue of which data values are critical. Ammann et al. [AJMB97] take a detailed look at the problem of surviving IW attacks on databases. They identify a number of phases of the IW process and describe activities which occur in each of them. They use a color scheme for marking damage and repair in databases and a notion of integrity suitable for databases that are partially damaged to develop a mechanism by which databases under attack could still be safely used.

In [JLM98], isolation is proposed as an IW defense mechanism that has been applied to protect systems from damage while investigating further. A scheme is described that isolates the database transparently from further damage by users suspected to be malicious, while still maintaining continued availability for their transactions. The interactions between the isolation component and other IW components such as the intrusion detector and the trusted recovery manager are also discussed.

As an earlier phase of trusted recovery (repair), *intrusion detection*, with the purpose of detecting a wide range of security violations ranging from attempted break-ins by outsiders to system penetrations and abuses by insiders, has attracted substantial research interests [Lun93, MHL94]. The methodology of intrusion detection can be divided into two categories: *anomaly detection* and *misuse detection.* Anomaly detection compares relevant data by statistical or other methods to representative profiles of normal, expected activity on the system or network. Deviations indicate suspicious behavior [JV94]. Misuse detection examines sniffer logs, audit data, or other data sources for evidence of operations, sequences, or techniques known to be used in particular types of attacks [Ilg93, GL91, PK92, IKP95, SG91, SG97, LWJ98]. Misuse detection techniques can not be used to detect new, unanticipated patterns that could be detected by anomaly detection techniques, but they perform better in detecting known attacks.

However, intrusion detection primarily focuses at the operating system level. Although work is ongoing to extend it to networks of distributed systems, it does not yet provide any help with intrusion detection at the level of DBMS. In a DBMS, the problem can be particularly difficult, since it involves detecting that data inserted into the database is unreasonable or incorrect. Although data jamming can be used to detect intruders, it usually can not be used to detect malicious transactions because the behavior of malicious transactions is just like the behavior of normal transactions which donot access dummy values.

Compared with other works in information warfare, this report differs in that it focuses on repair, as opposed to management, detection, protection, or availability, as cited above.

SECTION 3

Modelling the Underlying System

This chapter presents the foundation upon which a trusted recovery framework is built in later chapters. We explain what we mean by a database system and go into some details about our assumptions concerning database states, transactions, histories, and recovery models. We also explain what we mean by a database system that can survive IW attacks and go into some details about our assumptions concerning attacks, attack detection, and attack recovery.

3.1 Modelling Databases

In our framework, a database is specified as a collection of of data *items* (objects), along with some *invariants* or *integrity constraints* on these data items. At any given time, the database *state* is determined by the values of the items in the database. A change in the value of a data item changes the state. The integrity constraints are predicates defined over the data items. For example, in a banking system where a database is composed of a set of customer accounts, an integrity constraint over the database can be: 'the balance of each account must be greater than or equal to zero'. A database state is said to be *consistent* if the values of the data items satisfy the given integrity constraints. Otherwise, the state is inconsistent.

3.1.1 Transactions and Histories

A *transaction* is an execution of a program that transforms one database state to another. Associated with each transaction is a set of *preconditions* which limit the database states to which a transaction can be applied. A transaction is said to be *defined* on a database state if the state satisfies every precondition of the transaction.

From a more syntactic perspective, we model a transaction T_i as an ordered pair $(\sum_i, <_i)$, where $\sum_i$ is the set of operations in T_i, and $<_i$ indicates the execution order of those operations. A read (write) operation executed by a transaction T_i on item x is denoted as $r_i[x]$ ($w_i[x]$). Two operations *conflict* if one is write. We assume that there is at most one $r_i[x]$ and at most one $w_i[x]$ in $\sum_i$, and we further assume that if $r_i[x]$ and $w_i[x]$ are both in $\sum_i$, then $r_i[x] <_i w_i[x]$.

The primary purpose of a database management system (DBMS) is to carry out transactions. The traditional transaction model relies on the properties of *atomicity*, *consistency*, *isolation*, and *durability*. Atomicity ensures that the execution of a transaction is *atomic*, that is, a transaction either *commits* (denoted c_i), with all its changes being applied to the database, or *aborts* (denoted a_i), with all its changes being discarded. Consistency ensures that a transaction when executed by itself, without interference from other transactions, maps the database from one consistent state to another. Isolation ensures that no transaction ever views the partial effects of some other transaction even when transactions execute concurrently. Durability ensures that once a transaction successfully commits, all the state transformations of the transaction are made durable and public, even if there is a failure.

Transactions are usually executed *concurrently* for high performance. The execution of a set of transactions, denoted $\mathbf{T} = \{T_1, T_2, ..., T_n\}$, is modeled by a structure called a *history*. A history H over $\mathbf{T}$ is a partial order $(\sum, <_H)$, where $\sum$ is the set of all operations executed by transactions in $\mathbf{T}$, and $<_H$ indicates the execution order of those operations. Two histories are *conflict equivalent* if (1) they are defined over the same set of transactions and have the same operations, and (2) they order conflicting operations of nonaborted transactions in the same way.

The correctness of concurrent execution of transactions is typically captured by the notion of *serializability* [BHG87]. A history H is *serial* if, for any two transactions T_i and T_j that appear in H, either all operations of T_i appear before those of T_j or vice versa. A history H is *serializable* if its committed projection is conflict equivalent to a serial history. Serializable histories can be produced by many kinds of *concurrency control* protocols and *two-phase locking* (2PL) [BHG87], for example, is the most widely used concurrency control protocol in current database applications.

To ensure correctness in the presence of failures the DBMS must produce histories that are not only serializable but also *recoverable*. For an item x, we say that T_i *reads x from* T_j in history H if (1) $w_j[x] <_H r_i[x]$; (2) a_j does not precede $r_i[x]$ in $<_H$; and (3) if there is some $w_k[x]$ such that $w_j[x] <_H w_k[x] <_H r_i[x]$, then $a_k <_H r_i[x]$. We say that T_i *reads from* T_j in H if T_i reads some item from T_j in H. A history H is *recoverable* if, whenever T_i reads from T_j $(i \neq j)$ in H and $c_i \in H$, $c_j <_H c_i$. A history H is *strict* if whenever $w_j[x] <_H o_i[x]$, either $a_j <_H o_i[x]$ or $c_j <_H o_i[x]$ where $o_i[x]$ is $r_i[x]$ or $w_i[x]$.

3.1.2 A Database Recovery Model

Database recovery is the activity of ensuring that software and hardware failures, such as transaction failures, systems failures, storage media failures and communication failures, do not corrupt persistent data. We model a DBMS with a focus on recovery using the TM-scheduler-DM model which is proposed in [BHG87] and shown in Figure 3.1.

In the TM-scheduler-DM model, the *data manager* is in charge of recovery which consists of two components: a *cache manager* (CM), which provides operations to *fetch* data from stable storage into volatile storage, and to *flush* data from volatile to stable storage, and a *recovery manager* (RM) which processes Read, Write, Commit, Abort and Restart operations. In particular, the RM interface is defined by five procedures:

1. $RM - Read(T_i, x)$: read the value of x for transaction T_i;

2. $RM - Write(T_i, x, v)$: write v into x on behalf of transaction T_i;

3. $RM - Commit(T_i)$: commit T_i;

4. $RM - Abort(T_i)$: abort T_i; and

5. *Restart* : bring the stable database to the committed state following a system failure.

We assume the classical *write-ahead logging* (WAL) [MHL+92] is enforced in the TM-scheduler-DM model to ensure correct restarts. WAL requires that for each page the page's log records be flushed prior to overwriting its persistent (stable) copy. This is done by the CM under partial controls from the RM.

3.2 Modelling IW Attack and Defense

In order to degrade the confidentiality, integrity and availability of a database system, IW attack on the system can take many forms such as physical attack by destroying the stable storage (disks), Trojan Horse attack by providing the database administrator with malicious softwares, and operating system level attack by modifying database files without the interference of the DBMS. However, in order to concentrate on the problem of interest, we make the following two assumptions. Note that all the other kinds of attacks except malicious transactions are already well studied [Den83, GS96].

1. We assume that the behavior of the DBMS meets its specification, that is, there are no Trojan Horses in the system.

2. We assume that all IW attacks to the system are through malicious transactions, that is, there are no attacks bypassing the interference of the DBMS.

3.2.1 Model of a DBMS That Can Survive IW Attacks

We model a database system that can survive IW attacks with the architecture which is proposed in [JLM98] and shown in Figure 3.2.

As mentioned in Chapter 1, information warfare defense must consider the whole process of attack and recovery. This requires a recognition of the multiple phases of the IW process. The phases specified in Figure 3.2 and the activities that occur in each of them are as follows. Note that damage assessment and repair are the focus of this report.

Prevention: The defender puts protective measures into place. In particular, the *Policy Enforcement Manager* (PEM) enforces the access controls in accordance with the system security policy, such as MAC and DAC, on every access request. We assume no data access can bypass it. In addition, the system can be further protected by isolating the database transparently from further damage by users suspected to be malicious, while still maintaining continued availability for their transactions. This is achieved primarily by the *Isolation Manager.* Detailed discussion on isolation is out of the scope of this report.

Intelligence gathering: The attacker observes the system to determine its vulnerabilities and find the most critical functions or data to target. This phase is not directly specified in Figure 3.2.

Attack: The attacker carries out the resulting plan. In particular, the attacker first gets the required authorizations then issues some specific malicious transactions to the system (PEM).

Detection: The defender observes symptoms of a problem and determines that an attack may have taken place or be in progress. In particular, the *Intrusion Detection and Confinement Manager* applies either anomaly detection techniques, or misuse detection techniques, or both to identify suspicious behaviors as well as intrusions by malicious transactions. The detection is typically processed based on the information provided by the audit trail and/or the transaction logs.

Damage assessment: The defender determines the extent of the problem, including failed functions and corrupted data. In particular, when a malicious transaction is detected, the Intrusion Detection Manager notifies the *Damage Confinement and Assessment Manager* to confine and assess the damages caused by the transaction. The confinement can be done by notifying the PEM to restrict the following accesses of the user who issues the transaction, i.e., rejecting his/her further accesses. The assessment can be done by identifying which good transactions are affected by the malicious transaction and which data items are updated by either the malicious transaction or the affected good transactions.

Reconfiguration: The defender may reconfigure to allow operation to continue in a degraded mode while recovery proceeds. In particular, after the damages are assessed, the *Reconfiguration Manager* reconfigures the system to allow accesses to continue in a degraded mode while repair is being done by the *Damage Recovery Manager.* For example, the system can be continuously reconfigured to reject accesses to newly identified damaged data items and to allow accesses to newly recovered items.

Repair: The defender recovers corrupted or lost data and repairs or reinstalls failed system functions to reestablish a normal level of operation. In particular, after the Damage Assessment Manager informs which data items are damaged, and/or which good transactions are affected, the Damage Recovery Manager performs concrete repair algorithms, which will be developed in following chapters, to restore each damaged item to its cleaned value, and/or to remove the direct or indirect effects of the detected malicious transaction from the database. It should be noticed that in many situations damage assessment and recovery are coupled with each other closely. For example, damages can be recovered during the process of identifying and assessing damages. It should also be noticed that new malicious transactions can be detected during the process of assessing and recovering from the damages caused by older transactions.

Fault treatment: To the extent possible, the weaknesses exploited in the attack are identified and steps are taken to prevent a recurrence. This phase is not directly specified in Figure 3.2.

3.2.2 Detecting Malicious Transactions

As shown in Figure 3.2, taking place in an earlier phase of the information warfare countermeasures, detection (identification) of malicious transactions enables the processes of damage assessment and attack recovery. Although the effectiveness of damage assessment and recovery is heavily dependent on the effectiveness of malicious transaction detection, the techniques of damage assessment and recovery are almost independent of that of malicious transaction detection. This is also the reason why we can make the following assumption to enable us to concentrate on the techniques of damage assessment and recovery.

- We assume that the Intrusion Detection Manager can detect malicious transactions effectively.

In fact, there are many ways where such an identification (detection) could be specified. For example, all transactions associated with a specific user (i.e., an attacker), all transactions originating in a particular time window, or all transactions originating in an untrusted part of a network might comprise such a specification. Moreover, since the identification of an attacker may lead to the detection of a set of malicious transactions submitted by the attacker, traditional intrusion detection techniques for detecting malicious users [Lun93, MHL94, Den87] can be incorporated to do malicious transaction detection. Finally, more effective malicious transaction detection can be performed by exploiting transaction semantics. For example, in a banking system, from the same account a transaction withdrawing $10000 is usually more probable to be malicious than a transaction withdrawing $1000. Concrete malicious transaction identification mechanisms are outside the scope of this report.

3.2.3 Three Attack Recovery Models

The recovery methods that can be potentially enforced by the Damage Recovery Manager (see Figure 3.2) can be formalized around following three recovery models: HotStart, WarmStart, and ColdStart. HotStart is primarily a forward error recovery method, and ColdStart is primarily a backward error recovery method, but each of the three models incorporates both forward and backward error recovery to some degree. The three recovery models, HotStart, WarmStart, and ColdStart, are illustrated in Figure 3.3.

The HotStart model is appropriate for attacks where the system can or must respond transparently to the user. Suppose an attacker introduces a corrupt binary executable at a particular site and uses that executable to launch an availability, trust, or integrity attack. The attack can be handled with a HotStart model if two conditions hold. First, the attack must be detected early enough that damage is confined to the executable. Second, a hot standby of the executable - an uncorrupted standby, preferably at a different location - must be available to take over. The hot standby effects a recovery transparent to the user, even though the system is in a degraded state. It is still necessary to identify the path by which adversary introduced the corrupt binary, disable that path, and restore the proper binary from a back-up store.

Sometimes it is not possible to hide the effects of an attack from the users, and in these cases a WarmStart model is desirable. Damage can be confined such that key services are available, trustworthy, and reliable. Nonetheless, the user is aware of the attack because the system is visibly degraded. The exact level of service depends on the extent of the attack. Some functionality may be missing, untrustworthy, and/or based in incorrect information. Key mechanisms for managing WarmStarts are checkpoints for quick recovery and audit trails for intercepting the attacker.

The ColdStart model is appropriate for the most severe attacks. The chief difference from the WarmStart model is that the attacker succeeds in halting the delivery of system services. The goal of the ColdStart recovery is to bring the system back up as quickly as possible to a usable, trustworthy, and consistent state. Policies and algorithms are required to support efficient ColdStarts. Compensation for unrecoverable components - for example, leaked information - is also crucial.

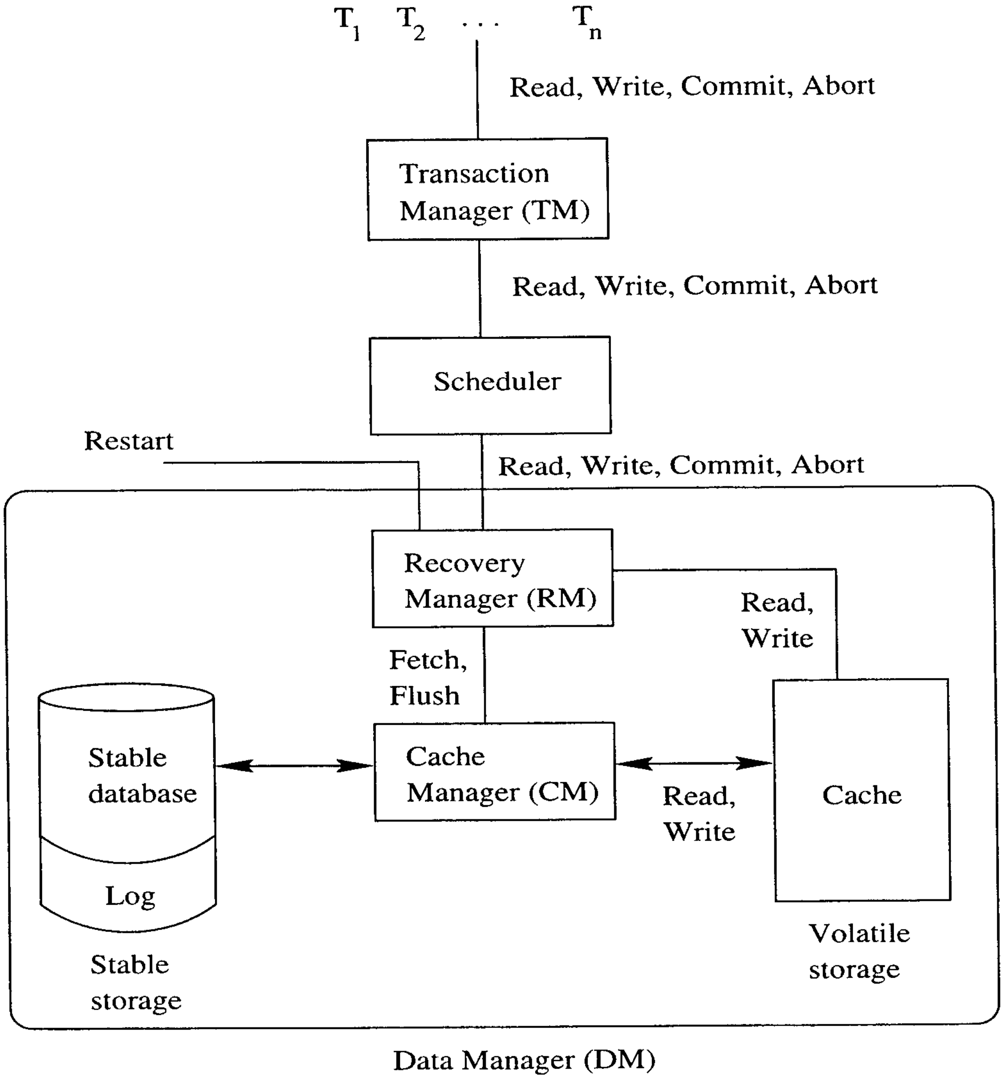


Figure 3.1: Model of a DBMS

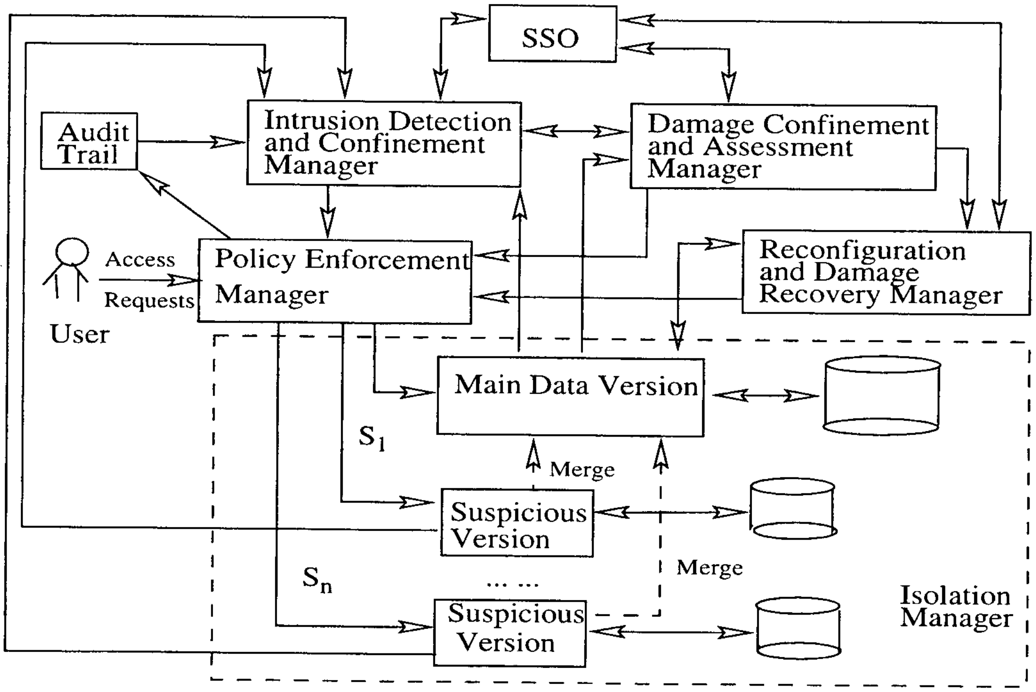

SSO
Intrusion Detection and Confinement Manager
Damage Confinement and Assessment Manager
Audit Trail
Access Requests
User
Policy Enforcement Manager
Reconfiguration and Damage Recovery Manager
Main Data Version
S1
Merge
Suspicious Version
Sn
... ...
Merge
Suspicious Version
Isolation Manager

Figure 3.2: Architecture of IW Defense

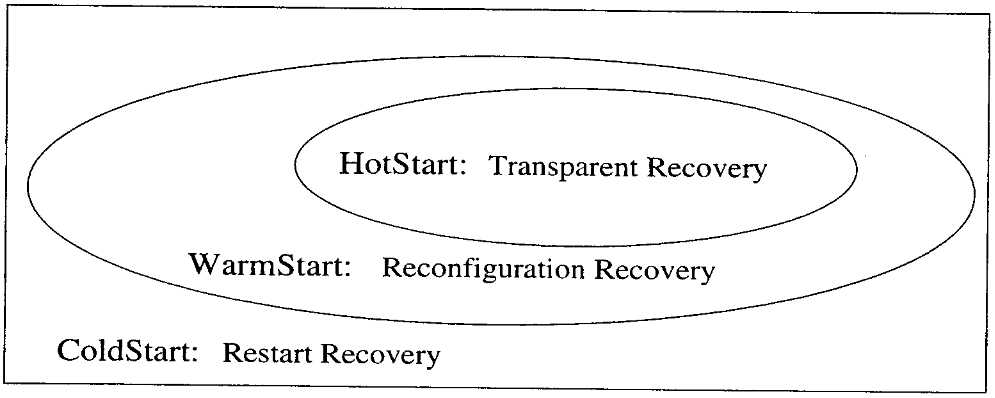

HotStart: Transparent Recovery
WarmStart: Reconfiguration Recovery
ColdStart: Restart Recovery

Figure 3.3: Recovery Models

SECTION 4

The Framework

This chapter presents a high-level framework within which trusted recovery can be supported and enforced. In particular, this chapter presents two recovery models to support 'undoing' undesirable committed transactions, such as malicious transactions and affected good transactions.

In previous presentation, we introduced the notation 'undo a committed transaction' because: (1) operations with the same operational semantics as traditional undos are needed to remove the effects of such committed transactions as malicious transactions and affected benign transactions, hence for simplicity we use the same word, namely 'undo', to denote such operations; and (2) it is very desirable to build our repair model on top of current DBMSes instead of building the model from scratch, because in this way current recovery mechanisms, such as undo and redo, can be directly exploited thus much more efficiency can be achieved, hence using notations consistent with traditional database recovery mechanisms is desired.

However, in traditional database systems 'undo' is a recovery mechanism that can only be applied to uncommitted transactions, and *durability*, a fundamental property of transaction processing, implies that there is no automatic function for revoking a committed transaction. Therefore, it is unclear, even confusing, to say 'undo a committed transaction'. Moreover, in traditional recovery 'undo a transaction' means removing all the changes of the transaction such that its effects will not be made durable and public to other transactions, however, in trusted recovery before a (committed) transaction is 'undone', its effects have already been made durable and disclosed to other transactions. Therefore, it is necessary to clarify the notation 'undo a committed transaction'. And it is desirable that we can extend classical recovery models (i.e., the model proposed in Figure 3.1) to do trusted recovery such that the durability property will not be violated.

For this purpose, this chapter presents two novel trusted recovery models to bridge the theoretical gap between classical database recovery theory and trusted recovery practice: (1) a *flat-transaction* recovery model where committed transactions are 'undone' by building and executing a specific

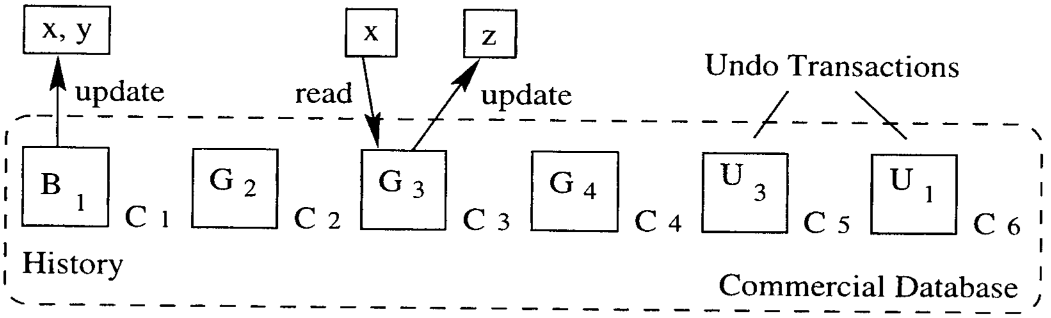


Figure 4.1: Flat Transaction Model

type of transactions, namely, *undo* transactions. It is simple and can fit in legacy system, but traditional undo facilities can not be directly exploited. (2) a *nested-transaction* model where a flat commercial history is virtually extended to a two-layer nested structure where originally committed transactions turn out to be subtransactions hence traditional undo operations can be directly applied to the model without violating the durability property. It fits in with legacy system, and inherently supports traditional undo.

4.1 Modelling Trusted Recovery by Flat Transactions

The flat transaction model is shown in Figure 4.1. This model has the durability property because committed transactions, no matter bad or good, will not be undone. Instead, for each committed transaction T_i which we want to 'undo', we build and execute a specific *undo transaction* (denoted U_i) to restore the data items updated by T_i to their before states. In other words, the notion 'undo a committed transaction T_i', in the flat transaction model, means the process of first building U_i then executing it.

To undo a committed transaction T_i, U_i is built as follows: for each update (write) operation of T_i, a write operation is appended to the program of U_i which writes the before value of the item updated. Therefore, U_i is composed of only write operations.

The execution of U_i is just like a normal transaction, for example, it can be executed concurrently with other normal or undo transactions. However, the effects of undoing a committed transaction T_i and the effects of undoing an uncommitted transaction T_j can be quite different. Undoing T_i removes the direct effects of T_i, that is, it restores every item updated by T_i, however it can not remove the indirect effects of T_i (if there are any), that is, it can not restore the items updated not by T_i but by some transactions affected by T_i. In contrast, undoing T_j can remove all the effects of T_j because the isolation property ensures that T_j can be backed out without affecting other transactions.

Figure 4.1 illustrates the idea. G_3 is affected by B_1 because G_3 reads x which is updated by B_1. It is clear that executing U_1 can restore items x and y to clean states, but after U_1 is executed item z can still stay in a dirty state because the value of z may have been affected by that of x read by G_3. Therefore, executing U_3 is required to remove the indirect effects of B_1. In practice, whether or not a system can be recovered from a malicious transaction B_i, that is, both of its direct and indirect effects can be removed, depends on whether or not we can identify these indirect effects accurately and execute the corresponding undo transactions in a proper order. These issues will be addressed in Chapter 5.

The advantage of the flat transaction model is that (1) it is simple, and (2) it fits in legacy system, not design from scratch. The drawback is that traditional undo operations need be enforced by building and executing undo transactions, thus classical undo facilities can not be directly exploited.

4.2 Modelling Trusted Recovery by Nested Transactions

The flat transaction model undos a committed transaction by executing a specific undo-transaction. Although undo-transactions are easy to build, traditional undo mechanisms can not be directly applied. It is desirable that we can exploit classical undo facilities directly without sacrificing performance objectives. We achieve this goal by proposing a nested transaction recovery model which is shown in Figure 4.2.

Consider a commercial database system where a history is composed of a set of (committed) flat transactions *, for a history to be repaired, we build the model by introducing a specific virtual transaction, called *malicious activity recovery transaction* (MART), on top of the history, and letting the MART be the parent of all the flat transactions in the history. As a result, the history is evolved into a nested structure where the MART is the top-level transaction and each flat transaction turns out to be a subtransaction whose execution is controlled by the MART.

Since in nested transaction processing [Mos85, LMWF94, GR93], subtransactions can theoretically be undone or compensated at anytime before the corresponding top-level transaction commits, so the model inherently supports undoing flat (commercial) transactions. This is also one of the reasons why we use the word 'undo' to denote one of our basic repair operations.

One interesting question about the model is 'Can a MART commit or abort, and, if this is possible, how to achieve this ?' It is clear that aborting the MART is equivalent to rolling back the

*Note that the recovery model can be easily extended to incorporate histories of multilevel or nested transactions.

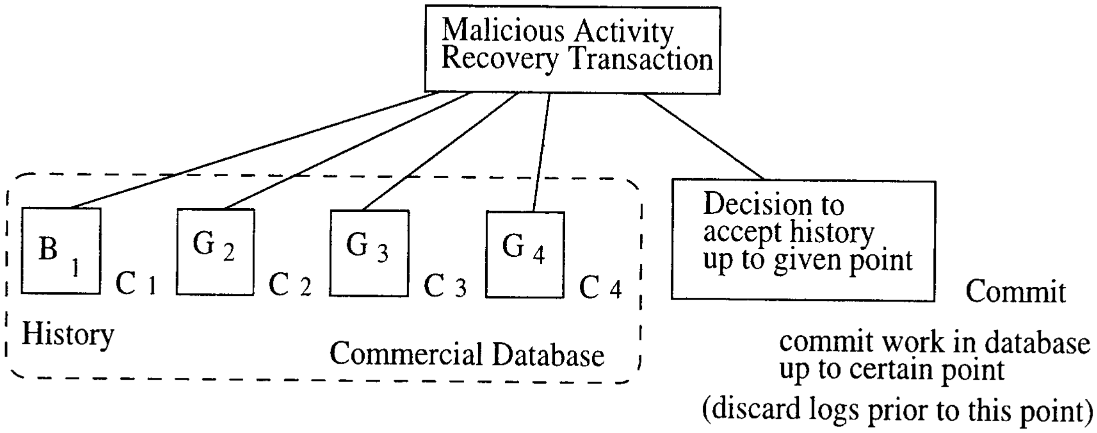

Malicious Activity Recovery Transaction
B 1
C 1
G 2
C 2
G 3
C 3
G 4
C 4
History
Commercial Database
Decision to accept history up to given point
Commit
commit work in database up to certain point
(discard logs prior to this point)

Figure 4.2: Nested Transaction Model

history to its initial state. However, how to commit MART is tricky. In fact, the MART should be able to be committed, because as the system keeps on executing new transactions the history can get tremendous long and the MART need to maintain too much information for the purpose of trusted recovery if the MART never commits. In practice, such information may no longer be available for a transaction T_i after T_i is committed for a long period of time. However, if we commit a MART at the end of the current history and start another new MART, then the work of some malicious transactions in the history supervised by the old MART can be committed before they are recovered. Hence we need to commit the work of good transactions while keeping the ability to recover from bad transactions. This goal is achieved by the following MART splitting protocol which is motivated by [PKH88].

- When the history is recovered to a specific point p_i, that is, it is believed that the effects of every bad transaction prior to p_i are removed, we can commit the work of all the transactions prior to p_i by splitting the MART into two MARTs: one supervising all the transactions prior to p_i, the other supervising the latter part of the history. Interested readers can refer to [PKH88] for a concrete process of transaction splitting which is omitted here.

- We commit the MART which supervises the part of the history prior to p_i. From the perspective of trusted recovery, the corresponding log records prior to p_i can be discarded to alleviate the system's resource consumption.

- We keep the other MART active so it can still be repaired.

It is clear that the nested recovery model fits in current commercial database systems very well thus trusted recovery need not be designed from scratch. First, each commercial flat transaction, as a subtransaction in the model, can be undone by directly applying traditional undo operations. In fact, we can see that in the model a savepoint is generated after each subtransaction commits so that the MART can rollback its execution to the beginning of any flat transaction. Second, as a conceptual framework, the model need never be really implemented to enable undoing committed (commercial) transactions. Therefore, the performance penalty caused by applying this model is very small.

The drawback of this model is that after a MART is committed there is no automatic ways to undo a flat transaction supervised by the MART even if the transaction is later identified as a malicious transaction. Therefore, from the perspective of trusted recovery, decisions to commit a MART should be made carefully .

SECTION 5

Trusted Recovery by Syntactic Approaches

After constructing in Chapter 3 the foundation of the proposed trusted recovery scheme and formalizing in Chapter 4 'undoing committed transactions', the fundamental trusted recovery operation, this chapter turns to address concrete trusted recovery mechanisms. In particular, this chapter presents the syntactic aspect of the proposed scheme where both coldstart and warmstart recovery algorithms are developed, but only syntactic dependencies between transactions are exploited.

In this chapter, if there is no specific clarification, the operation 'undo a committed transaction' can be understood as being modeled either by the flat transaction model or by the nested transaction model.

5.1 The Model

Assumptions

We assume that the histories to be repaired are serializable histories generated by some mechanism that implements a classical transaction processing model [BHG87]. We denote committed undesirable or *bad* transactions in a history by the set $\mathbf{B} = \{B_1, B_2, ..., B_m\}$. We denote committed desirable or *good* transactions in a history by the set $\mathbf{G} = \{G_1, G_2, ..., G_n\}$. Since recovery of uncommitted transactions is addressed by standard mechanisms, we consider a history H over $\mathbf{B} \cup \mathbf{G}$.

Transaction Dependencies

One simple repair is to roll back the history until at least the first bad transaction, and then try to reexecute all of the undone good transactions. The drawback of this approach is that many good transactions may be unnecessarily undone and reexecuted. Consider the following history over (B_1, G_1, G_2):

$H_1 : r_{B_1}[x] w_{B_1}[x] c_{B_1} r_{G_1}[y] r_{G_2}[x] w_{G_1}[y] c_{G_1} w_{G_2}[x] c_{G_2}$

It is clear that G_1 need not be undone and reexecuted since it does not conflict with B_1. We formalize the notion that some - but not all - good transactions need to be undone and reexecuted in the usual way:

Definition 1 Transaction T_j is *dependent upon* transaction T_i in a history if there exists a data item x such that:

1. T_j reads x after T_i has updated x;
2. T_i does not abort before T_j reads x; and
3. every transaction (if any) that updates x between the time T_i updates x and T_j reads x is aborted before T_j reads x.

Every good transaction that is dependent upon some bad transaction needs to be undone and reexecuted. There are also other good transactions that also need be undone and reexecuted. Consider the following history over (B_1, G_1, G_2):

$H_2 : r_{B_1}[x] w_{B_1}[x] c_{B_1} r_{G_1}[x] w_{G_1}[x] r_{G_1}[y] w_{G_1}[y] c_{G_1} r_{G_2}[y] w_{G_2}[y] c_{G_2}$

G_2 is not dependent upon B_1, but it should be undone and reexecuted, because the value of x which G_1 reads from B_1 may affect the value of y which G_2 reads from G_1. This relation between G_2 and B_1 is captured by the transitive closure of the dependent upon relation:

Definition 2 In a history, transaction T_1 *affects* transaction T_2 if the ordered pair $(T_1,\ T_2)$ is in the transitive closure of the *dependent upon* relation. A good transaction G_1 is *suspect* if some bad transaction B_1 affects G_1.

It is convenient to define the *dependency graph* for a set of transactions S in a history as $DG(S) = (V, E)$ in which V is the union of S and the set of transactions that are affected by S. There is an edge, $T_i \rightarrow T_j$, in E if $T_i \in V$, $T_j \in (V - S)$, and T_j is dependent upon T_i. Notice

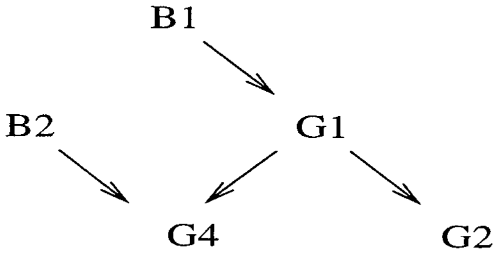


Figure 5.1: Dependency Graph for History H_3

that there are no edges that terminate at elements of S; such edges are specifically excluded by the definition. As a result, every source node in $DG(\mathbf{B})$ is a bad transaction, and every non-source node in $DG(\mathbf{B})$ is a suspect transaction.

As an example, consider the following history over $(B_1, B_2, G_1, G_2, G_3, G_4)$:

$$\begin{aligned} H_3 : {} & r_{B_1}[x]w_{B_1}[x]c_{B_1}r_{G_1}[x]w_{G_1}[x]r_{G_3}[z]w_{G_3}[z]c_{G_3}r_{G_1}[y]w_{G_1}[y]c_{G_1} \\ & r_{G_2}[y]w_{G_2}[y]r_{B_2}[z]w_{B_2}[z]c_{B_2}r_{G_2}[v]w_{G_2}[v]c_{G_2}r_{G_4}[z]w_{G_4}[z]r_{G_4}[y]w_{G_4}[y]c_{G_4} \end{aligned}$$

$DG(\mathbf{B})$ is shown in Figure 5.1.

If a good transaction is not affected by any bad transaction (for example, G_3 in H_3), then the good transaction need not be undone and reexecuted. In other words, only the transactions in $DG(\mathbf{B})$ need be undone, and only the suspect transactions in $DG(\mathbf{B})$ need to be reexecuted. From the recovery perspective, the goal is to first get $DG(\mathbf{B})$, then undo all these transactions.

Before we continue, we modify our model with respect to blind writes.* We developed the model as is because it captures exactly the set of suspect transactions that must be undone, assuming that further information about the transactions - such as data flow or semantic information - is unavailable. Specifically, the model includes an optimization for blind writes. Suppose a transaction in **B** writes x and subsequently a good transaction blindly overwrites x. Then the *dependent upon* chain is broken, and other good transactions that subsequently read x will not necessarily appear in $DG(\mathbf{B})$.

From the perspective of the recovery algorithms developed in this report, we view this optimization as counterproductive for two reasons. First, blind writes are relatively infrequent in many applications. Second, accommodating blind writes would complicate the recovery algorithms we present. We make the design decision that the optimizations of blind writes are not worth the additional storage and processing time that would be required in the algorithms. To accommodate

*A transaction blindly writes a data item if it writes a value without first reading the item.

this decision, for the remainder of this report we assume that transactions do not issue blind writes. That is, if a transaction writes some data, the transaction is assumed to read the value first.

We say a data item x is *dirty* if x is in the write set of any bad or suspect transaction. From the data perspective, the goal is to restore each dirty data item to the value it had before the first transaction in $DG(\mathbf{B})$ wrote it. The resulting state will appear as if the bad and suspect transactions never executed.

It is clear that the dependency graph of history H can not be built without the corresponding read information for transactions in H. Unfortunately, the read information we can get from the logs for traditional recovery purposes such as physical logs, physiological logs, and logical logs [GR93], is usually not enough for constructing the $DG(\mathbf{B})$. Therefore, the efficient maintenance of read information is a critical issue. In particular, there is a tradeoff between the extra cost we need to pay besides that of traditional recovery facilities and the guaranteed availability of read information.

There are several possible ways to maintain and capture read information. For example,

- augment the write log to incorporate read information.
- extract read sets from the profiles of transactions.
- extract read information from physiological or logical logs.
- build an online dependency graph.

Based on the amount of available read information provided by these methods, we can achieve several types of repair:

- A repair is *complete*, if the effects of every bad or suspect transaction are repaired.
- A repair is *exact*, if the effects of all and only bad or suspect transactions are repaired.

Since the specification and the properties of our repair algorithms are closely related to the approach which is selected to maintain the read information, we present the algorithms in a way which is based on the different read information capturing methods, although the basic ideas of these algorithms are very similar. The coldstart as well as the warmstart repair algorithms based on In-Log read information are introduced in Section 5.2 and Section 5.3 respectively. The repair algorithms based on the read information extracted from transaction profiles are specified in Section 5.4.

5.2 Static Repair Based on In-Log Read Information

Our basic repair algorithm is based on traditional recovery mechanisms [BHG87]. One advantage of this approach is that we need not develop the repair algorithm from scratch. In addition, the standard recovery mechanisms need not be modified greatly to accommodate the repair algorithm.

We use the same physical log as used in traditional recovery mechanisms [BHG87] except that we define a new type of log record to document every read operation. These records are used to construct the dependent upon relation between transactions. The read log record $[T_i, x]$ denotes that the data item x is read by transaction T_i. An algorithm that does not modify the log, but instead maintains the read log separately, is discussed in section 5.2.3. As mentioned in Chapter 3, we use the same TM-Scheduler-DM model of centralized database systems as used in [BHG87]. We add one action, which appends $[T_i, x]$ to the log, to the RM-Read(T_i, x) procedure. We assume that the scheduler invokes RM operations in an order that produces a serializable and strict execution.

The basic idea of static repair is that we halt the processing of transactions periodically after a set of bad transactions B is identified, and then we build the $DG(\mathbf{B})$, based on the log and/or other available read information, to identify the bad as well as the suspect transactions.

5.2.1 Three Pass Repair Algorithm

The algorithm described below is composed of three passes. Pass one scans the log forward from the entry where the first bad transaction commits to produce a list of all the transactions which commit after the first bad transaction. Some good transactions in this list may be suspect. Pass two scans the log forward from the entry where the first bad transaction starts and extracts every bad and suspect transaction from the commit list of pass one. Pass three goes backward from the end of the log to undo all bad and suspect transactions.

Algorithm 1 Three Pass Repair Algorithm
Input: the log, the set **B** of bad transactions.
Output: A consistent database state in which all bad and suspect transactions are undone.
Initialization.

```
Let $commit\_list = \{\}$, $undo\_list = \{\}$, $write\_set = \{\}$, $tmp\_write\_set = \{\}^{(A)}$.
Pass 1.
1. Locate the log entry where the first bad transaction $B_1$ commits.
2. Scan forward until the end of the log. For each log entry,
2.1 if the entry is a commit record $[T_i, commit]$
        $commit\_list = commit\_list \cup \{T_i\}$;
```

Pass 2.

```
1. Locate the log entry where the first bad transaction $B_1$ starts.
2. Scan forward until the commit_list turns empty or there are no more log entries to examine. For each log entry,
2.1 if the entry is for a transaction $T_i$ in B
        if $T_i$ is not in the undo_list
            $undo\_list = undo\_list \cup \{T_i\}$;
        if the entry is a write record $[T_i, x, v]$
            $write\_set = write\_set \cup \{x\}$;
2.2 elseif the entry is for a transaction in the commit_list
        case the entry is a write record $[T_i, x, v]$
            if $T_i$ is in the undo_list
                $write\_set = write\_set \cup \{x\}$;
            else $tmp\_write\_set = tmp\_write\_set \cup \{(T_i, x)\}^{(B)}$;
        case the entry is a read record $[T_i, x]$
            if $T_i$ is in the undo_list
                skip the entry;
            elseif $x$ is in the write_set
                $undo\_list = undo\_list \cup \{T_i\}$ ;
                move all the data items of $T_i$ from the tmp_write_set to the write_set;
        case the entry is a commit record $[T_i, commit]^{(C)}$
            if $T_i$ is not in the undo_list
                delete all the data items of $T_i$ from the tmp_write_set;
```

Pass 3.

Scan backward from the end of the log to undo all the transactions in the *undo_list*.

Comments

A. The *commit_list* consists of the transactions which commit after the first bad transaction. The *undo_list* consists of the bad and suspect transactions that should be undone. The *write_set* consists of the dirty data items. The use of *tmp_write_set* is explained in comment B.

B. When we encounter a write log entry for a good transaction which is still not in the *undo_list*, we cannot be sure whether the transaction will become suspect - it may read some dirty data later on. At this time, we need to keep track of the data items written by this transaction in case we have to add them to the *write_set* later. There are basically two approaches to solve this problem. One, which is used in the algorithm, is to keep the write items in a temporary memory structure (namely, *tmp_write_set*); the other is to scan the log backward to figure out the write set later on (the backward scan can be efficient since all the write log entries of a transaction are chained together in the log). The first approach costs more memory space but is faster. The second approach costs less memory space but is slower since it may cause disk operations.

Also, since we assume that the history to be repaired is strict, the following scenario, which happens in a history that is recoverable but not strict, will not occur:

$r_{G_1}[x_1]w_{G_1}[x_1]r_{G_2}[x_1]w_{G_2}[x_1]r_{G_1}[y_1]c_{G_1}c_{G_2}$

Suppose y_1 is in the *write_set*, x_1 and x_2 are not. When we encounter the entry $r_{G_2}[x_1]$, though G_2 is dependent upon G_1 we skip it according to the algorithm since x_1 is not in the *write_set*. Later when we encounter the entry $r_{G_1}[y_1]$, we will add G_1 to the *undo_list* since it reads an item in the *write_set*. But at this point, G_2 will not be added to the *undo_list* though it has been affected by G_1.

C. This step improves the performance of the algorithm when the history is long and when many committed good transactions are not affected.

Theorem 1 Given the state produced by history H over $\mathbf{B} \cup \mathbf{G}$, Algorithm 1 constructs the state that would have been produced by H', where H' is H with all transactions in $DG(\mathbf{B})$ removed.

Proof: Given the relationship between *dirty* data, the *bad* and *suspect* transactions, this theorem amounts to showing that each dirty data item is restored to the latest value before it turns dirty. The following three claims are sufficient to show this.
Claim 1. Every bad and suspect transaction is added to the *undo_list* in Pass 2. It is clear that every bad transaction is added to *undo_list* in step 2.1 of Pass 2. Suppose there are some suspect transactions which have not been added to the *undo_list* and T_i is the first one. Then according to the algorithm, it happens that when T_i reads a dirty item x_i in step 2.2 of Pass 2 x_i is still not in the *write_set*. Since the execution is strict, when T_i reads x_i every bad or suspect transaction that writes x_i before the read operation has already committed, and therefore x_i is already added to the *write_set* in step 2.1 or 2.2 of Pass 2, which contradicts with the assumption.

Claim 2. Only bad and suspect transactions are added to the *undo_list* in Pass 2. Suppose some non suspect good transactions have been added to the *undo_list* and T_i is the first one. Then according to the algorithm, it happens that when T_i reads an item x_i in step 2.2 x_i is in the *write_set*. Therefore T_i *is* formally suspect, which contradicts the assumption.

Suppose x_i is a dirty data item and T_i is the first transaction which makes x_i dirty, then T_i must be either bad or suspect.

If T_i is the first bad transaction, then x_i will be restored to the value before T_i writes it in pass three. If T_i is not, then no bad or suspect transaction that commits before T_i has updated x_i because otherwise T_i cannot be the first transaction that makes x_i dirty. By *Claim 1* , we know that every bad or suspect transaction that commits before T_i is considered here, and we know T_i will be added to the *undo_list* in pass 2. By *Claim 2* , there is no non suspect good transaction which updates x_i, commits before T_i and is in the *undo_list*. Thus x_i will be restored to the before image of T_i's write which is the latest un-dirty value of x_i. T_i cannot be the first transaction that makes x_i dirty. □

5.2.2 Two Pass Repair Algorithm

One drawback of the three pass repair algorithm is that it needs three passes, which may take too much time if the log is large. At the cost of additional memory, the first two of these passes can be combined. In two pass algorithm, the forward pass gets the list of all bad and suspect transactions; the backward pass undoes these transactions. For brevity, and to highlight the differences between the two algorithms, we describe only the modifications to the three pass algorithm.

```
Algorithm 2 Two Pass Repair Algorithm
Omit pass 1 of Algorithm 1.
In pass 2 of Algorithm 1 add another list tmp_undo_list to capture the set of in-repair good trans-
actions which have read some dirty data.† For each entry which is not for a transaction in B:
    case the entry is a write record $[T_i, x, v]$
        $tmp\_write\_set = tmp\_write\_set \cup \{(T_i, x)\}$;
    case the entry is a read record $[T_i, x]$
        if $x$ is in the write_set
            $tmp\_undo\_list = tmp\_undo\_list \cup \{T_i\}$;
    case the entry is an abort record $[T_i, abort]$
        delete all the data items of $T_i$ from the tmp_write_set;
        if $T_i$ is in the tmp_undo_list
            delete $T_i$ from the tmp_undo_list;
    case the entry is a commit record $[T_i, commit]$
        if $T_i$ is in the tmp_undo_list
            move $T_i$ from the tmp_undo_list to the the undo_list;
```

† A transaction T is *in-repair* between the time we scan the record $[T, begin]$ and the time we scan the record $[T, commit]$ or the record $[T, abort]$.

```
        move all the data items of $T_i$ from the *tmp_write_set* to the *write_set*;
    **else** delete all the data items of $T_i$ from the *tmp_write_set*;
```

Theorem 2 The two pass algorithm and the three pass algorithm are equivalent, in the sense that for any input they get the same output database state.

Proof: It is clear that any aborted good transaction will not be added to the *undo_list* in both algorithms. For a committed suspect transaction T_i, it will be added to the *undo_list* in the three pass algorithm when it reads an item in the *write_set*, and some data items written by T_i may be added to the *write_set* before T_i commits. In the two pass algorithm, T_i will not be added to the *undo_list* until it commits, so does the data items written by T_i. Since the history is strict, any data item written by T_i will not be read by other transactions until T_i commits. Therefore, the different time in these two algorithms when T_i is added to the *undo_list* and when T_i's data items are added to the *write_set* will not influence the output. So these two algorithms are equivalent. □

5.2.3 Repair Algorithm Based on Separate Read Log

The three pass and the two pass algorithms are based on the log to which read records are added. Sometimes, it is desirable to use a separate log to document the read operations rather than change the traditional log. We call the separate log *read log*, and we call the traditional log *update log*. Using a read log to repair a history has the advantage that the traditional recovery mechanisms do not have to be modified to take a different data structure for the log into account.

There is only one type of entry of the form $[T_i, x]$ in the read log, identifying the data item x which is read by transaction T_i.

Conceptually, a two pass repair algorithm based on the read log as well as the update log can be designed using the same memory data structure and algorithm as used in Algorithm 2 if we can transform the serial scan operations in Algorithm 2 over one log to some equivalent interleaved scan operations over the read log and the update log. Thus, one important issue in using a read log to do repair is to synchronize the scan operations over the update log and the read log.

The order by which we interleave the scan operations over the update log and the read log is critical to the correctness of the repair algorithm. If an entry $[T_i, x]$ in the read log is scanned earlier than an entry in the update log which denotes an operation happening before the read, then T_i may not be added to the *undo_list* though it is a suspect transaction. Look at the following scan sequence:

$r_{G_1}[x_1]r_{G_1}[y_1]w_{G_1}[y_1]r_{G_2}[y_1]w_{G_1}[x_1]c_{G_1}w_{G_2}[y_1]c_{G_2}$

Suppose x_1 is in the *write_set*. When we scan the entry $r_{G_2}[y_1]$, we cannot find that G_2 reads a dirty item since y_1 is not in the *write_set* yet. Later on, when we scan the entry c_{G_1}, we will add y_1 to the *write_set* but G_2 will be found not to be suspect since it will not read y_1 again. The point is that $r_{G_2}[y_1]$ happens after c_{G_1} (since the execution is strict) but it is scanned before c_{G_1}.

If an entry $[T_i, x]$ in the read log is scanned later than an entry in the update log which denotes an operation happening after the read, then we may not find the write items of T_i in the *tmp_write_set* when we find T_i suspect since all the write items of T_i may have been deleted. Look at the following scan sequence:

$w_{G_1}[x_1]w_{G_1}[y_1]c_{G_1}r_{G_1}[x_1]r_{G_1}[y_1]$

Suppose x_1 is in the *write_set*. When we scan the entry c_{G_1}, we will delete all the write items of $G_1(x_1,\ y_1)$ from the *tmp_write_set* since G_1 has not read any dirty data. Later on, when we encounter the entry $r_{G_1}[x_1]$, we find G_1 is suspect but we cannot find the items written by G_1 from the *tmp_write_set*.

So we must synchronize the scan operations in a way which can ensure the correctness of the algorithm. The requirement implied by this can be conveniently stated as two design rules that every two pass repair algorithm which uses read logs must observe.

Rule 1: Before a read entry $[T_i, x]$ is scanned in the read log, any write record for x which denotes an operation happening before the read must have been scanned.

Rule 2: Before we scan a commit record $[T_i, commit]$ in the update log, all the read records for T_i must have been scanned.

Our synchronizing mechanism is specified as follows.

Mechanism 1 When a read entry is added to the read log, the largest *LSN* [BHG87] of the update log will be recorded in the read entry. For an entry r_1 in the update log and an entry r_2 in the read log, let $r_1.LSN$ denote the LSN of r_1, let $r_2.read\_LSN$ denote the LSN recorded in r_2, r_1 and r_2 are scanned in the following order:

1. If $r_2.read\_LSN \geq r_1.LSN$, then r_1 is scanned before r_2.

2. If $r_2.read\_LSN < r_1.LSN$, then r_2 is scanned before r_1.

Lemma 1 Mechanism 1 ensures that the scan order of r_1 and r_2 is the order in which the operations denoted by r_1 and r_2 happen. And the mechanism satisfies the two design rules.

Proof: Let o_1 be the operation denoted by r_1(o_1 may be a commit, abort, or update operation); let o_2 be the read operation denoted by r_2. When o_2 happens, we create r_2 and record the largest *LSN* of the update log in the *read_LSN* field of r_2. Since at that time the operation o_p denoted by the entry of the update log with the *LSN* has already happened, o_p happens before o_2. Since o_p denotes the last operation logged in the update log before o_2 happens, every operation which is logged in the update log later than o_p happens after o_2. Therefore, if $r_2.read\_LSN \geq r_1.LSN$, then $o_p.LSN \geq r_1.LSN$, so o_1 is o_p or happens before o_p, thus o_1 happens before o_2. If $r_2.read\_LSN < r_1.LSN$, then $o_p.LSN < r_1.LSN$, so o_1 happens after o_p, thus o_1 happens after o_2. Therefore our scan order is the operation order. Since the operation order satisfies the two design rules, Mechanism 1 satisfies the two design rules. □

The repair algorithm based on separate read log is described as follows:

Algorithm 3 Repair Algorithm Based on Separate Read Log
Use Mechanism 1 to schedule the order in scanning the update log as well as the read log. For every kind of log entry, do the same thing as Algorithm 2. For brevity, the details are omitted.

Theorem 3 The algorithm based on separate read log and the two pass algorithm are equivalent, in the sense that for any input, they get the same output database state.

Proof: Since Algorithm 3 uses Mechanism 1, it scans the entries in the update log as well as the read log in the same order as the two pass algorithm scans the traditional log associated with the records for read operations. In addition, for every entry, Algorithm 3 does the same thing as Algorithm 2. Therefore, these two algorithms are equivalent. □

5.3 On-the-Fly Repair Based on In-Log Read Information

The three pass algorithm, the two pass algorithm, and the algorithm based on separate read log which we have presented in Section 5.2 are all static repair, or coldstart, methods. New transactions are blocked during the repair process. In some database applications, availability requirements dictate that new transactions be able to execute concurrently with the repair process, that is, the application requires warmstart semantics for recovery. The cost of on-the-fly repair is that some new transactions may inadvertently access and subsequently spread damaged data.

The traditional transaction management architecture is adequate to accommodate on-the-fly repair (see Figure 5.2) [‡].

[‡] Note that Figure 5.2 is adapted from Figure 3.1. For simplicity, here we omit the TM and incorporate the function of the CM in the RM.

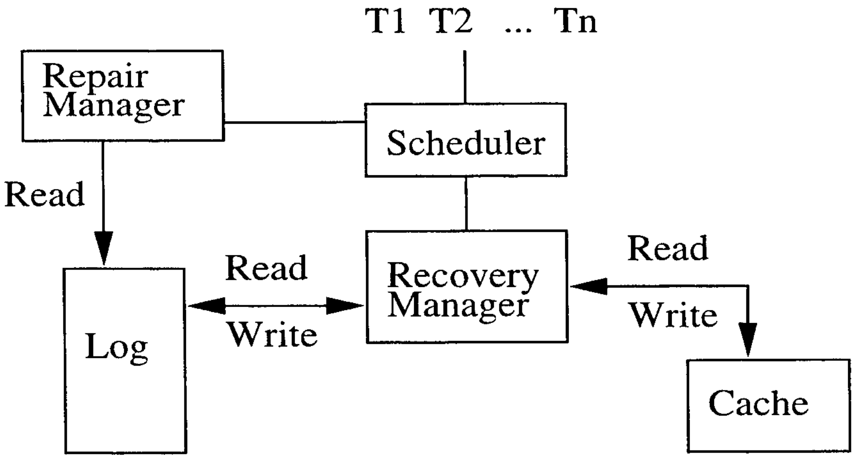


Figure 5.2: Architecture of the on-the-fly repair system

The *Repair Manager* is applied to the growing logs of on-the-fly histories to mark any bad as well as suspect transactions. For every bad or suspect transaction, the Repair Manager builds an undo transaction and submits it to the *Scheduler* [§]. The undo transaction is only composed of write operations.

The *Scheduler* schedules the operations submitted either by user transactions or by undo transactions to generate a correct on-the-fly history. Suspect transactions that are undone can be resubmitted to the Scheduler either by users or by the Repair Manager.

The *Recovery Manager* executes the operations submitted by the Scheduler and logs them. It keeps the read information of transactions either in a traditional log modified to store read operations or in a separate read log.

For simplicity in the presentation, we assume that each new transaction is good.

5.3.1 Termination Detection

New transactions are continuously submitted to the Scheduler, and as a result, the log keeps growing. A key question is 'Does repair terminate, and if so, is termination detectable?'

Suppose at some point the Repair Manager has repaired the history up to record a on the log (See Figure 5.3). That is, every bad or suspect transaction which commits before a is logged has been undone, its dirty data items have been marked and cleaned. Suppose record b is the present

[§] Note that here the flat transaction model is used.

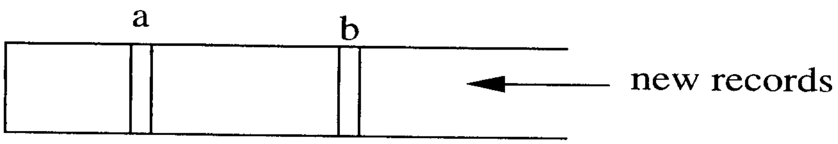


Figure 5.3: A Snapshot of Repair on the Log

bottom of the log. It is possible that a newly submitted read operation reads a dirty item which has not been marked, because the item can be made dirty by some write operation which happened between a and b. Since neither the Scheduler nor the Repair Manager can detect this, the read operation is not rejected. In this way, some newly submitted good transaction may become suspect. As an example, consider the following operation sequence:

$$r_{G_{i1}}[x_1]w_{G_{i1}}[x_2]c_{G_{i1}}r_{G_{i2}}[x_2]w_{G_{i2}}[x_3]c_{G_{i2}}...r_{G_{ik}}[x_k]w_{G_{ik}}[x_{k+1}]c_{G_{ik}}...$$

Even if x_1 is the only dirty data item when the sequence begins, repair may not terminate until the submission terminates because when x_i is cleaned, x_{i+1} may already become dirty.

Consider another operation sequence:

$$G_{i1}G_{i2}...G_{i(k-1)}r_{G_{ik}}[x_1]w_{G_{ik}}[x_2]c_{G_{ik}}...$$

Assume that only x_1 is dirty when the sequence begins and none of the transactions between G_{i1} and $G_{i(k-1)}$ read x_1. Then it is possible that when G_{ik} reads x_1, every bad or suspect transaction has already been repaired. Thus, x_1 is clean, and the repair terminates.

Whether or not a repair terminates depends on the repair speed, the arrival rate of new transactions, and the nature of the new transactions. So, in general, termination of repair cannot be guaranteed without taking additional measures, which are discussed later. However, if the repair process is complete, this condition can be detected. We turn to termination detection next.

Checking if every marked dirty data item has been cleaned to determine if repair is complete is not sufficient for two reasons. First, some transaction T which has been found suspect may write dirty data items later on (see Figure 5.4): at time t_5 the read record $[T, x]$ is scanned and T is found suspect since x was dirty when T read x (Notice that when $[T, x]$ is scanned x may not be dirty since x may already be cleaned at t_4); at time t_6 every dirty item that is marked before t_6 has been cleaned, but the repair does not terminate since at time t_7, T writes an item y and y becomes dirty. Second, some transaction which has not yet been identified as suspect may generate dirty data (See Figure 5.5): $[T, begin]$ record is scanned after time t_4 when no data is dirty, we can not stop repair at time t_4 since at time t_6 we find T is suspect and at time t_7 item y is marked dirty.

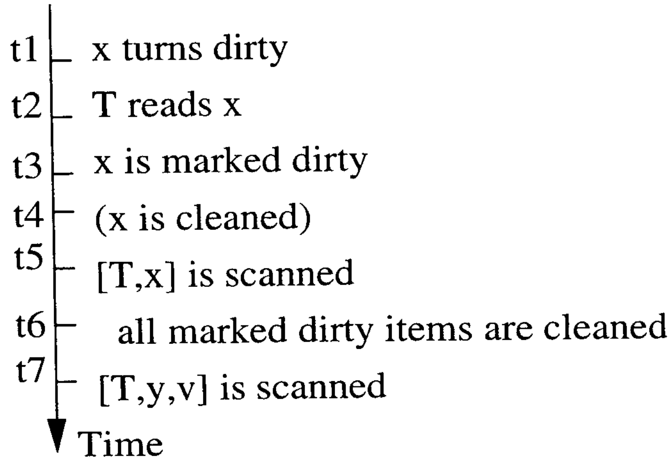


Figure 5.4: Transactions which have been found suspect may generate new dirty items

From another perspective, when data item x is read or written, x may be at one of the seven kinds of states denoted in Figure 5.6.

Before x turns dirty, x is in the 'clean' state (state 1). x is in the 'pseudo clean' state (state 2) between the time x turns dirty and the time x is marked dirty. x is in the 'dirty' state (state 3) between the time x is marked dirty and the time x is cleaned. x is in the 'cleaned' state (state 4) between the time x is cleaned and the time x turns dirty again. x is in the 'pseudo cleaned' state (state 5) between the time x turns dirty again and the time x is marked dirty again. x is again in the 'dirty' state (state 6) between the time x is marked dirty again and the time x is cleaned again. x is again in the 'cleaned' state (state 7) between the time x is cleaned again and the time x turns dirty again. Of course, these states may be repeated indefinitely.

Mechanism 2 described below can capture the two situations shown in Figure 5.4 and Figure 5.5, thus can detect the termination of On-the-fly repair processes.

Mechanism 2 In the process of repair:

- Maintain a *dirty_item_set* to keep every data item in state 3 or 6; maintain a *cleaned_item_set* to keep every data item in state 4 or 7. We show how to capture these items in Section 5.3.4.

- Associate each item x in the *cleaned_item_set* with a number, $x.LSN$, which denotes the log serial number of the bottom record of the log at the time when x is cleaned.

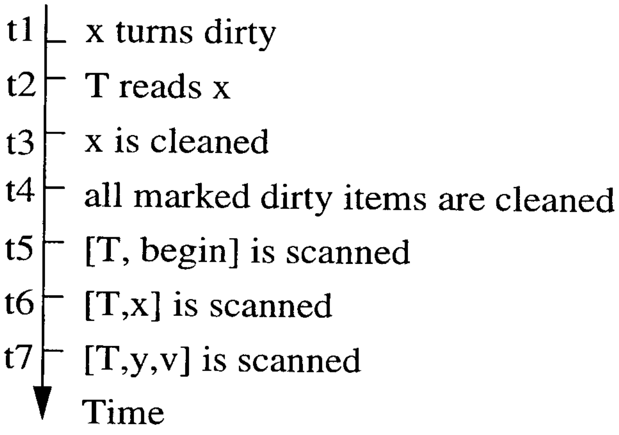


Figure 5.5: Transactions which will later be found suspect may generate new dirty items

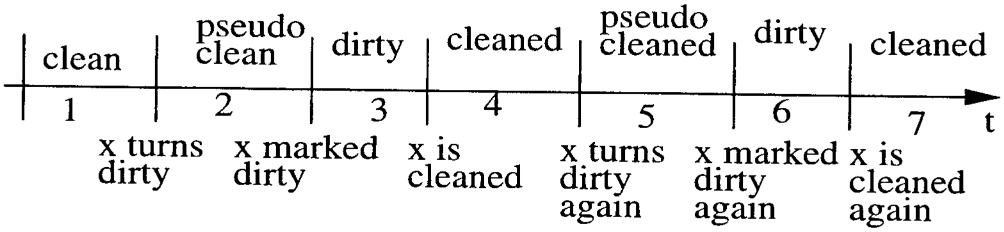


Figure 5.6: Possible Item States

- Maintain a *tmp_undo_list* to keep every in-repair transaction that has read some data item in the *dirty_item_set*, or has read an item x in the *cleaned_item_set* where $r.LSN \leq x.LSN$. Here $r.LSN$ is the log serial number of the read record.
- We report that the repair terminates if
  - every bad transaction in **B** has been undone, and
  - $dirty\_item\_set = \emptyset$, and
  - $tmp\_undo\_list = \emptyset$, and
  - $\forall x \in cleaned\_item\_set$, $x.LSN < l.LSN$. Here, $l.LSN$ denotes the log serial number of

the next log record for the Repair Manager to scan.

Theorem 4 Mechanism 2 reports termination iff the repair process, in fact, terminates.

Proof: Repair terminates iff all the marked dirty items have been cleaned and it is not possible for any item to turn dirty later on. When Mechanism 2 reports termination every marked dirty item has been cleaned since *dirty_item_set* $= \emptyset$. At this time, since every bad transaction in **B** has been undone, an item x may turn dirty later on only if x is written by a suspect transaction which has been detected or by a suspect transaction which will be detected later on.

An transaction T can be found suspect only if there is an item x such that T read x when x was dirty. When $[T, x]$ is scanned, x may still be dirty or may have been cleaned, but x can not be first cleaned and then marked dirty for the following reason. Suppose the transaction that makes x dirty again is T'. Then the write record $[T', x, v]$ can only be scanned after $[T, x]$ since it is appended to the log after $[T, x]$. Therefore, when $[T, x]$ is scanned x is still dirty, and so x must be in the *dirty_item_set*. If x has been cleaned, then $r.LSN \leq x.LSN$. So every transaction that has been found suspect will be in the *tmp_undo_list*. Therefore, when *tmp_undo_list* $= \emptyset$ no such transaction exist.

When *dirty_item_set* $= \emptyset$ an item x will be written by a transaction T which will be found suspect later on only if T had read x before x is cleaned, but when T reads x, x is still dirty. (This situation is shown in Figure 5.5.) When Mechanism 2 reports termination, $\forall x \in$ *cleaned_item_set*, $x.LSN < l.LSN$, that is, every dirty item is cleaned before the operation denoted by the next log record for the Repair Manager to scan, Therefore, every read operation denoted by a record that the Repair Manager is going to scan will not read any dirty item, so the situation will not happen. □

5.3.2 Building Undo Transactions

On-the-fly repair requires the Repair Manager build and submit the undo transactions for every bad or suspect transaction, that is, the Repair Manager starts to built the undo transaction for a transaction T_i as soon as T_i is found bad or suspect. Since the log keeps on growing, the undo can only be done from the beginning to the end of the history, which is different from the methods presented in Section 5.2.

The straight forward way to build undo transactions for bad or suspect T_i is to scan backward along the log from the point where T_i commits, and for every write record of T_i, add a corresponding write operation to the undo transaction U_i. The write operation in U_i restores the item to its old value. This approach does not work if new transactions execute concurrently with repair. Consider the event sequence shown in Figure 5.7. If x is clean before T_i writes x, T_i's undo transaction U_i

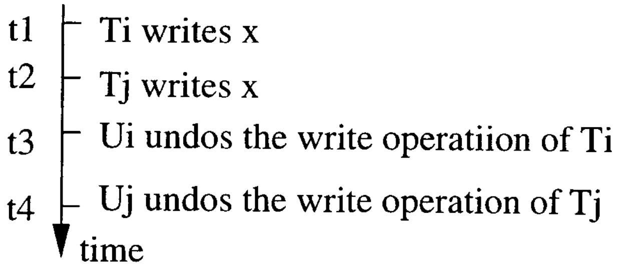

t1
Ti writes x
t2
Tj writes x
t3
Ui undos the write operatiion of Ti
t4
Uj undos the write operation of Tj
time

Figure 5.7: The flaw of the straight forward method

undos this write operation at time t_3 and x is cleaned. However, the undo transaction U_j of another suspect transaction T_j undos the write operation of T_j on x at time t_4 and x turns dirty again, which is not correct.

Algorithm 4 described below fills the hole of the straight forward method.

Algorithm 4 Building Undo Transactions

1. Maintain a *submitted_item_set* to keep every item x whose undo operation has been submitted to the Scheduler, but x still has not been cleaned.

2. When building an undo transaction, for every write record which is scanned, if the record is on an item x which is in the *cleaned_item_set* or in the *submitted_item_set* then omit the record; if x is in the *write_item_set* but not in the *submitted_item_set*, then add the corresponding undo operation to the undo transaction and add x to the *submitted_item_set*.

Theorem 5 In Algorithm 4, when U_i is built, every dirty data item x of T_i will either be repaired in a operation of U_i or in a operation of another undo transaction, and x will be restored to the value x had before it turned dirty.

Proof: If x is clean or cleaned before T_i writes x, then the undo operation $w_{U_i}[x]$ will restore x to the latest value before x turned dirty. If x is dirty before T_i writes x, suppose T_j is the transaction which makes x dirty, then when $[T_i, x, v]$ is scanned, x is either cleaned, so in the *cleaned_item_set*, or is submitted by the undo transaction which is built for T_j, so in the *submitted_item_set*, therefore, U_i will not repair x. □

5.3.3 On-the-fly Concurrency Control

Before introducing the On-the-fly repair algorithm, we need to first analyze how the Scheduler should schedule the user operations as well as the undo operations to achieve repair.

To define the acceptable histories generated by the Scheduler, we associate the read and undo operations in histories with appropriate states of the Repair Manager (i.e., the state of the *dirty_item_set*) when the operations are scheduled to execute, and use the states to indicate the correctness of repair histories.

Definition 3 History H is a *correct on-the-fly history* if

1. H is serializable and strict,
2. There are no abort records for undo transactions,
3. For any read operation $r_{T_i}[x]$, the predicate $x \notin$ *dirty_item_set* holds,
4. For any conflicting undo transaction pair U_i and U_j, if $T_i <_H T_j$ then $U_i <_H U_j$, and
5. For any undo operation $w_U[x]$, the predicate $(x \in$ *dirty_item_set*$) \cap (x \in$ *submitted_item_set*$)$ holds.

Statement 3 says that when a read operation $r_{T_i}[x]$ is scheduled x must be clean or cleaned. Statement 4 says that conflicting undo transactions should be scheduled in the same order in which they are submitted by the Repair Manager (As shown in Section 5.3.2, the order is critical to the correctness of repair.). Statement 5 says that when an undo operation $w_U[x]$ is scheduled, x must be dirty.

The scheduling algorithm is described as follows:

Algorithm 5 Scheduling Algorithm
The algorithm is based on strict 2PL. The modification lies in:

- Never abort undo transactions;
- When a read operation $r_{T_i}[x]$ arrives, if x is in the *dirty_item_set*, then reject this read operation and rollback T_i.

We show in next section that if the Scheduler executes Algorithm 5, then together with the Repair Manager, and the Recovery Manager, the Scheduler generates correct on-the-fly histories.

An important task of the Scheduler is to control the submitting speed of user operations so that the repair can eventually finish. Informally, the Repair Manager can slow down the submitting speed of user operations when the repair process fails to terminate in a satisfactory time frame. An automatic way to control the speed is as follows. Periodically the Scheduler evaluates the trend in the size of the *dirty_item_set*. The trend can be captured with time series analysis techniques. If the trend is up, then the submitting speed can be reduced. Otherwise, termination is on track.

5.3.4 On-the-fly Repair Algorithm

The integrated On-the-fly repair algorithm consists of three parts which are executed on the Repair Manager, the Scheduler, and the Recovery Manager, respectively.

Algorithm 6 On-the-fly Repair Algorithm
Input: the log, the set **B** of bad transactions.
Output: if the repair terminates at the middle of the history, then any prefix H_p of the history including the point where the repair terminates results in the state that would have been produced by H'_p, where H'_p is H_p with all transactions in $DG(\mathbf{B})$ removed. If the repair terminates at the end of the history H, then H will result in the state that would have been produced by H', where H' is H with all transactions in $DG(\mathbf{B})$ removed.
Initialization:
Let *tmp_undo_list* = {}, *cleaned_item_set* = {}, *dirty_item_set* = {}, *tmp_item_set* = {}.
At the Repair Manager:

```
1. Locate the log entry where the first bad transaction $B_1$ starts.
2. while (the termination conditions do not hold [¶])
      Scan next log entry:
        if the entry is for a transaction $T_i$ in B
           if the entry is a write record $[T_i, x, v]$ and $x$ is not in the cleaned_item_set
              add $x$ to the dirty_item_set;
           if the entry is a commit record $[T_i, commit]$
              build the undo transaction for $T_i$ using Algorithm 4 and
              submit it to the Scheduler;
        else
           case the entry is a write record $[T_i, x, v]$
              if $x$ is not in the cleaned_item_set
                 add $x$ to the tmp_item_set;
              elseif $w.LSN > x.LSN^{(A)}$
                 add $x$ to the tmp_item_set;
```

[¶] The termination conditions are stated in Mechanism 2.

```
        case the entry is a read record $[T_i, x]$
          if $x$ is in the dirty_item_set or
          $x$ is in the cleaned_item_set and $r.LSN \leq x.LSN$
            add $T_i$ to the tmp_undo_list;
        case the entry is an abort record $[T_i, abort]$
          delete all the data items of $T_i$ from the tmp_item_set;
          if $T_i$ is in the tmp_undo_list, remove it;
        case the entry is a commit record $[T_i, commit]$
          if $T_i$ is in the tmp_undo_list
            move all the items of $T_i$ from the tmp_item_set to the dirty_item_set;
            build the undo transaction for $T_i$ using Algorithm 4 and submit it
            to the Scheduler;
          else delete all the items of $T_i$ from the tmp_item_set;
3. report termination; exit;
```

At the Scheduler:

Schedule the user operations as well as the undo operations using Algorithm 5.

At the Recovery Manager:

When an undo operation $w_{U_i}[x]$ is done, delete item x from both the *dirty_item_set* and the *submitted_item_set*, then add $(x, x.LSN)$ to the *cleaned_item_set.*

Comments

A. $w.LSN$ denotes the log serial number of the write record. Notice that when $w.LSN > x.LSN$ x is cleaned before the write operation, therefore, the write operation may make x dirty again. Otherwise, x is cleaned after the write operation, so x will not be made dirty again by this operation, therefore x need not be cleaned anymore.

Theorem 6 Algorithm 6 meets its specification.

Proof: Given the relationship between *dirty* data, the *bad* and *suspect* transactions, this theorem amounts to showing that at the time when the repair terminates each dirty data item is restored to the latest value before the data item turns dirty.

Claim 1. The Scheduler generates only correct on-the-fly histories. From the definition of 2PL and Algorithm 5, we know that the first three statements of Definition 3 hold. The Repair Manager builds and submits undo transactions in the scanning order, and before an undo operation $w_{U_i}[x]$ is executed and x is cleaned any conflicting undo operation $w_{U_j}[x]$ will not be submitted to the Scheduler. This is because between the time $w_{U_i}[x]$ is submitted and the time it is executed any newly submitted user transaction which reads x will be rejected, and the Repair Manager will not build any other undo operation to repair x. Therefore, statements 4 and 5 hold.

Claim 2. Algorithm 6 realizes Mechanism 2, and thus reports termination correctly.

Claim 3. In the Repair Manager, at any point of time every dirty data item x in the part of the history having been scanned by the Repair Manager has been marked and the corresponding undo operation, which can restore the value of x to the latest value before x turns dirty, has been built and submitted to the Scheduler. Since in the part of history, an item x can be first made dirty, then cleaned, and then made dirty again, we associate a dirty item x with the period of time when it remains dirty(denoted p). Thus (x, p_1) and (x, p_2) denote two different dirty items. As shown in Algorithm 6, for every dirty data item (x, p) an undo operation and only one undo operation will be built to repair it at the very beginning of p. See Theorem 1 and Theorem 4 for the reason that every dirty data item is marked. □

5.4 Extracting Read Information From Transaction Profiles

Sections 5.2 and 5.3 detail recovery algorithms that, given a specification of malicious, committed transactions, unwind the effects of each malicious transaction, along with the effects of any benign transaction that depends, directly or indirectly on a malicious transaction. The significance is that the work of the remaining benign transactions is saved. However the assumption that read information are kept in the log may incur substantial performance penalties due to the significant storage and processing cost of maintaining read information.

There are basically two ways to keep read information in the write log or in another read log. One way is what we assumed in Sections 5.2 and 5.3, that is, let the RM-Read(T_i, x) procedure append the read record $[T_i, x]$ to the log every time when T_i reads an item x. The other way is to let the RM-Read(T_i, x) procedure keep the set of items read by T_i in another place until the time when T_i is going to commit, at this point, the read set of T_i can be put into the log as one record. Compared with the first approach, the second approach saves some storage since the identifier of T_i need not be put into the log repeatedly, however, it may require the database to store relatively large data objects because read sets can be very big. In addition, it may delay termination detection during a warmstart repair process [‖].

[‖] The corresponding coldstart and warmstart repair algorithms based on the second approach are the same as the algorithms presented in this section.

Although keeping read information in the log will not cause more forced I/O, it does consume more storage. Though the previous two approaches need to keep only the identifier instead of the value of each read item in the log, the size of a read set can still be very big. For example, in a bank a transaction which generates the monthly statement of a customer needs to read the information of every transaction submitted by the customer during the last month.

Another problem with keeping read information in logs lies in the fact that almost all present database systems keep only update(write) information in the log. Adding read records to the log may cause the redesign of the current recovery mechanisms, including both the data structure and the algorithms.

Any way of maintaining read information should keep the malicious transaction recovery module isolated from the traditional recovery module as much as possible. Such an approach avoids degrading the performance of the traditional recovery module and also makes it easier to build the malicious transaction recovery module on the top of the existing database systems.

In this section, we adopt the approach of extracting read information from the profiles and input arguments of transactions. Compared with the read log approach, each transaction just needs to store its input parameters, which are often much smaller in size than the read set. More important, instead of putting the input parameters in the log, each transaction can store the parameters in a specific user database, thus the damage recovery module can be completely isolated from the traditional recovery module. In this way, our repair model can be easily implemented on the top of current database systems without any change to the DBMSs. The only thing we need to do is to let application programmers change the transaction code such that damage recovery can be supported. The approach is not exact, and as a result, it may back out some non-suspect good transactions and/or delay termination detection during a warmstart repair process.

5.4.1 The Model

We start with the transaction profile of TPC-A, a well known database benchmark [Gra93], as an example. TPC-A is stated in terms of a hypothetical bank. The bank has one or more branches. Each branch has multiple tellers. The bank has many customers , each with an account. The database represents the cash position of each entity(branch, teller, and account) and a history of recent transactions run by the bank. The transaction represents the work done when a customer makes a deposit or a withdrawal against his account. The transaction profile is specified as follows:

```
Input: Aid, Tid, Bid, Delta
BEGIN TRANSACTION
   Update Account_Balance where Account_ID = Aid:
      Read Account_Balance from Account
```

```
        Set Account_Balance = Account_Balance + Delta
        Write Account_Balance to Account
    Write to History:
        Aid, Tid, Bid, Delta, Time_stamp
    Update Teller where Teller_ID = Tid:
        Set Teller_Balance =  Teller_Balance + Delta
        Write Teller_Balance to Teller
    Update Branch where Branch_ID = Bid:
        Set Branch_Balance =  Branch_Balance + Delta
        Write Branch_Balance to Branch
COMMIT TRANSACTION
```

Here, Aid(Account_ID), Tid(Teller_ID), and Bid(Branch_ID) are keys to the relevant records(rows). For this transaction, the read set in tuple(record) level is:

Read_Set= { Account.Aid, Teller.Tid, Branch.Bid}

Each item in the set uniquely identifies a tuple of a relation. At the element level, the read set is:

Read_Set= { Account.Aid.Account_Balance, Teller.Tid.Teller_Balance,
Branch.Bid.Branch_Balance }

Each data item is an element of a relation. In this example, the item is composed of three parts, namely the relation identifier, *Account*, the record identifier, *Account.Aid*, and the attribute identifier, *Account_Balance*. To find the record identified by *Account.Aid*, the DBMS usually needs to search the corresponding index. However, we do not put searching keys such as *Account.Aid.Account_ID* into the read set because we assume that the primary key of a relation is not updated unless the record is deleted.

5.4.2 Read Set Templates

As shown in the example above, given the source code and the input arguments, it is possible to extract exact or approximate read sets from transactions. However extracting read sets on the fly, that is, analyzing transaction source code during execution, may not meet the requirement of current online transaction processing systems. The reason is that extracting read set can cause an unacceptable processing delay.

An efficient method of getting read sets is required. Since every transaction running in a OLTP system typically belongs to some category, we assume that a transaction type is associated with every transaction, which identifies the nature of the transaction. Transactions of the same type are the same program, though they typically execute with different inputs.

The *read set template* for a transaction type is a representation of the data items that will be read by transactions of the type. Since read set templates are generated based on only transaction profiles, there are no real input arguments available and each data item in a read set template can only be specified as a function of the input variables.

An efficient way to extract read information from transaction profiles based on read set templates is as follows:

1. Analyze the source code of each type of transaction offline and get the read set template of that type.

2. When a transaction T is submitted to the Scheduler, the read set template for T's type is materialized with the input arguments of T. The process of materializing is done by substituting each variable in the read set template with its corresponding real value.

3. The materialized read set template is the read set for T.

For example, there is only one type of transaction in TPC-A, its read set template is:

Template = { Account.Aid.Account_Balance; Teller.Tid.Teller_Balance;
Branch.Bid.Branch_Balance }

For a transaction instance with the input $(Aid =' A1591749', Tid =' T0002', Bid =' BGMU001', Delta =$ $1000), the read set for the transaction is:

Read_Set= { Account.$'A1591746'$.Account_Balance, Teller.$'T0002'$.Teller_Balance,
Branch.$'BGMU001'$.Branch_Balance }

As shown in the above example, for any TPC-A transaction instance and for any database state on which the transaction is executed, we can get the *exact* read set based on its read set template, that is, the materialized template will indicate all and only the data items which are read by the transaction, either in tuple level or in element level. However, in some circumstances based on a template we may only get an approximate read set.

For example, in the **Stock-Level** transaction of TPC-C [Gra93], we retrieve the stock level of the last 20 orders of a district from the table **Order** and the table **Stock**. The numbers of these orders are traced from the D_NEXT_O_ID field of the district record in the table **District** which is identified by the input *w_id+d_id* ('+' denotes string concatenation). The read set template can be specified as follows:

Template= { x = District.(w_id+d_id).D_NEXT_O_ID;
$R_1 = \{x-1, ..., x-19, x-20\}$;
R_2 = Order-Line.(w_id+d_id+R_1+OL_NUMBER).OL_I_ID;
Stock.(w_id+R_2).S_QUANTITY }

Here, R_1 is the set of the numbers of the last 20 orders. Order-Line.(w_id+d_id+ R_1) identify the order lines for the 20 orders whose numbers are kept in R_1. In the record identifier part of each item, low case words denote variables which can be traced from the input; capital words denote attributes or sets of variables. The attributes, i.e., OL_NUMBER, can take any value. Based on the transaction profile, we can trace the D_NEXT_O_ID field from the input, however we can not trace further from R_1 to the last 20 orders because the value of x depends on the concrete database state when the transaction is executed.

There are two approaches to materialize the template. One is *generalizing*, that is, to view R_1 as the set of all order numbers, thus the template can be materialized by only the input. The other is *tracing*, that is to materialize R_1 based on the database context, for example, when doing repair we can scan back from the point of the log where the transaction was executed to get the value of x. The second approach, though can achieve finer repair, may cause substantial extra costs, especially in dynamic repair scenarios.

Besides *exact* read sets, *potential* read sets maintain approxiate read items for transactions. That is, for each item in the potential read set of transaction T, there exists a database state under which the item will be read by T when it is executed. It is clear that the potential read set for a transaction is the union of all the possible exact read sets of the transaction. Since we materialize read set templates before transactions are executed, and since we do not predict control flows within transaction, in some cases we only get potential read sets, and not exact read sets.

Since only database objects can be put into read set templates, we focus on the DML statements which play as the interface between transactions and databases. **Insert** statements add new tuples to relations, thus will not bring new read items; **Delete** statements replace database items with null values, thus will not add new read items. Therefore, only **Select** and **Update** statements introduce new read items.

For a **Select** or **Update** statement s of a transaction T, the input of s is the values(may denoted as variables) which are used in the **Where** or **Set** clauses of s. The input may come directly from the input of T, or indirectly from some previous query or program statement. It is clear that every template extraction must satisfy the following properties:

- For each **Select** statement, the template can not be larger than the union of all the relations in its **From** clauses. For each **Update** statement, the template can not be larger than the union of all the relations in its **Update** and **From** clauses.

- For each transaction, the template can not be larger than the union of all the templates for every **Select** or **Update** statement.

- The data items in the template for transaction T depend only on the transaction program, and not on any particular database state.

The read set templates of transactions can be extracted in three steps:

Step 1. Extract the template for each **Select** or **Update** statement separately.

Step 2. Combine the template for each **Select** or **Update** statement to get the template for the transaction.

Step 3. Generalize the template as appropriate.

For example, there are two **Select** statements in the **Stock-Level** transaction. In step 1, the template for the first statement is:

TP_1 = { District.(w_id+d_id).D_NEXT_O_ID }

The template for the second statement is:

TP_2 = { R_1 ={ o_id-1, ..., o_id-19, o_id-20 };
R_2 = Order-Line.(w_id+d_id+R_1).OL_I_ID;
Stock.(w_id+R_2).S_QUANTITY }

In step 2, based on the relation between TP_1 and TP_2 that o_id = District.(w_id+d_id). D_NEXT_O_ID, we get the combined template which is specified in the above example.

In situations where tracing through the log for the value of some variable in the template does not justify the corresponding cost, a simpler template materialized from only the input is preferred. This is done in Step 3. For this example, the generalized template is:

Template= { District.(w_id+d_id).D_NEXT_O_ID;
Order-Line.(w_id+d_id+OL_O_ID+OL_NUMBER).OL_I_ID;
Stock.(w_id+R_1.S_QUANTITY }

Based on the different possible structures of a **Select** or **Update** statement, some rules can be followed in Step 1:

affecting rule: If data item d_1 affects d_2, then d_1 should be put into the template so long as d_2 is in the template.

set rule : If the result of the statement is based on a funtion of a set of database objects, then the whole set should be put into the template.

join rule: If the result of the statement is got from the join of several relations, then every join key, except primary keys, should be in the template.

mapping rule: Map the aggregate functions in the statement to the corresponding set operations in the template. Map nested **Select** statements to nested templates. Map **Exist** clauses to the emptiness judgement of nested templates. Map **Views** to the corresponding SQL statements of the views.

Based on the different possible control flows in a transaction program, some rules can be followed in Step 2:

branching rule: For branching program units, such as **if-else** and **case**, with the standard form **if** c **then** $SS1$ **else** $SS2$, assume the read set templates for $SS1$ and $SS2$ are RS_1 and RS_2 respectively, then the read set template for the unit is: **if** c **then** RS_1 **else** RS_2.

loop rule: Viewing a loop structure as a limited set of program blocks, the read set template of the loop statement is the union of the templates of all its member blocks.

Templates can be generalized based on the following rule:

container rule: For any data item x which is read by transaction T, if x can be directly specified by the input of T, that is, no database state is needed in the specification, then add x into the template. Otherwise, add the least set of data items which includes x and can be directly specified by the input into the template. The least set of items is called the *container* of x.

5.4.3 Examples

In this section, we show as a feasibility exercise some realistic transaction examples from which the read set templates can be extracted. We adopt the transaction examples from benchmarks for transaction processing systems [Gra93].

TPC-B

The transaction profile of TPC-B is almost the same as TPC-A, so they have the same read set Template.

TPC-C

The benchmark portrays a wholesale supplier with a number of geographically distributed sales districts and associated warehouses. As the Company's business expands, new warehouse and associated sales districts are created. Each regional warehouse covers 10 districts. Each district serves 3000 customers. All warehouse maintain stocks for the 100,000 items sold by the Company. Customers call the Company to place a new order or request the status of an existing order. Orders are composed of an average of 10 order items.

In TPC-C, the term **database transaction** as used in the specification refers to a unit of work on the database with full ACID properties. A **business transaction** is composed of one or more database transactions. In TPC-C a total of five types of business transactions are used to model the processing of an order (See [Gra93] for the source codes of these transactions.).

- The **New-Order** transaction consists of entering a complete order through a single database transaction. The template for this type of transaction is:

  Input= warehouse number(w_id), district number(d_id),
  customer number(c_id); a set of items(ol_i_id),
  supplying warehouses(ol_supply_w_id), and quantities(ol_quantity).
  Read_Set= { Warehouse.w_id.W_TAX;
  District.(w_id+d_id).(D_TAX, D_NEXT_O_ID);
  Customer.(w_id+d_id+c_id).(C_DISCOUNT, C_LAST,
  C_CREDIT);
  Item.ol_i_id.(I_PRICE, I_NAME, I_DATA);
  Stock.(ol_supply_w_id+ol_i_id).(S_QUANTITY, S_DIST_xx,
  S_DATA, S_YTD, S_ORDER_CNT, S_REMOTE_CNT) }

- The **Payment** transaction updates the customer's balance, and the payment is reflected in the district's and warehouse's sales statistics, all within a single database transaction. The template for this type of transaction is:

  Input= a warehouse number(w_id), district number(d_id),
  customer number(c_id) or customer last name(c_last),
  and payment amount(h_amount)
  Read_Set= { Warehouse.w_id.(W_NAME, W_STREET_1, W_STREET_2,
  W_STATE, W_YTD);
  District.(w_id+d_id).(D_NAME, D_STREET_1, D_STREET_2,
  D_CITY, D_STATE, D_ZIP, D_YTD);

[**Case 1,** the input is customer number:
Customer.(w_id+d_id+c_id).(C_FIRST, C_LAST,C_STREET_1,
C_STREET_2, C_CITY, C_STATE, C_ZIP, C_PHONE,
C_SINCE, C_CREDIT, C_CREDIT_LIM, C_DISCOUNT,
C_BALANCE, C_YTD_PAYMENT, C_PAYMENT_CNT,
C_DATA);
Case 2, the input is customer last name:
Customer.(w_id+d_id+c_last).(C_FIRST, C_LAST,C_STREET_1,
C_STREET_2, C_CITY, C_STATE, C_ZIP, C_PHONE,
C_SINCE, C_CREDIT, C_CREDIT_LIM, C_DISCOUNT,
C_BALANCE, C_YTD_PAYMENT, C_PAYMENT_CNT,
C_DATA)] }

- The **Order-Status** transaction queries the status of a customer's most recent order within a single database transaction. The template for this type of transaction is:

Input= a customer number(w_id+d_id+c_id) or
customer last name(w_id+d_id+c_last)
Read_Set= { [**Case 1,** the input is customer number:
Customer.(w_id+d_id+c_id).(C_BALANCE,C_FIRST, C_LAST,
C_MIDDLE);
Case 2, the input is customer last name:
Customer.(w_id+d_id+c_last).(C_BALANCE,C_FIRST, C_LAST,
C_MIDDLE)] ;
x=Order.(w_id+d_id+c_id).O_ID;
Order.(w_id+d_id+c_id).(O_ENTRY_D, O_CARRIER_ID);
Order-line.(w_id+d_id+x).(OL_I_ID, OL_SUPPLY_W_ID,
OL_QUANTITY, OL_AMOUNT, OL_DELIVERY_D) }

- The **Delivery** transaction processes ten new (not yet delivered) orders within one or more database transactions. The template for this type of transaction is:

Input= a warehouse number(w_id), district number(d_id),
and a carrier number(o_carrier_id)
Read_Set= { x = New-Order.(w_id+d_id).NO_O_ID;
y = Order.(w_id+d_id+x).O_C_ID;
Order.(w_id+d_id+x).(O_CARRIER_ID,

OL_DELIVERTY_D, OL_AMOUNT);
Customer.(w_id+d_id+y).(C_BALANCE, C_DELIVERY_CNT) }

- The **Stock-Level** transaction determines the number of recently sold items that have a stock level below a specified threshold. The template for this type of transaction is specified in the previous presentation.

AS3AP

In AS3AP, since the performance is tested by separated pieces of codes, such as *selections*, *joins*, *projections*, *aggregations*, and *updates*, every transaction is so simple that the read set templates can be easily built.

Set Query Benchmark

The types of transactions in the benchamrk are: Q1, Q2A, Q2B, Q3A, Q3B, Q4, Q5, Q6A, Q6B. Since these transactions are relatively simple, it is easy to build the read set templates for them.

5.4.4 Static Repair

Using the read information extracted from transaction profiles, the static repair algorithms, either the three pass algorithm, or the two pass algorithm, can be modified correspondingly to achieve the goal.

For the three pass algorithm, the modified version is:

Algorithm 7 Three Pass Repair Algorithm
Input: the log, the set **B** of bad transactions.
Output: a consistent database state in which all bad and suspect transactions are undone.
Pass 1. scan the log from the beginning to the end to get the serial order of only the committed transactions.
Pass 2. scan the log along the serial order got in pass 1; for each transaction in **B** add its write items to the set *dirty_set*; for each good transaction, if the intersection of its *read set* and the *dirty_set* is not empty, then add its write items to the set *dirty_set* and mark it 'suspect'.
Pass 3. undo all the bad transactions as well as the marked suspect transactions.

The modified version of two pass algorithm is:

Algorithm 8 Two Pass Repair Algorithm
Input: the log, the set **B** of bad transactions.
Output: a consistent database state in which all bad and suspect transactions are undone.
Pass 1. scan the log from the beginning to the end; for each transaction in **B** add its write items to the set *dirty_set*; for each good transaction keep its write items until the *commit* or *abort* record is scanned; if it commits, and the intersection of its *read set* and the *dirty_set* is not empty, then add its write items to the set *dirty_set* and mark it 'suspect'.
Pass 2. undo all the bad transactions as well as the marked suspect transactions.

We note that these two algorithms are both based on the assumption that the scanning order of transactions is a serial order, because the write-read dependency is based on the serial order. Fortunately, strict two phase locking, used in most commercial systems, ensures that the commit order is the serial order if read locks are not released before a transaction commits or aborts.

5.4.5 Dynamic Repair

On-the-fly Concurrency Control

The same as Algorithm 5.

Correct On-the-fly history

A history is *read-strict* if it is strict, and if whenever $r_j[x] < o_i[x] (i \neq j)$, either $a_j < o_i[x]$ or $c_j < o_i[x]$ where $o_i[x]$ is $r_i[x]$ or $w_i[x]$. That is, no data item may be read or overwritten until the transaction that previously read or wrote into it terminates, either by aborting or by committing.

Definition 4 History H is a *correct on-the-fly history* if

1. H is serializable and read-strict,
2. There is no abort records for undo transactions,
3. For any read operation $r_{T_i}[x]$, the predicate $x \notin \mathit{dirty\_item\_set}$ holds,
4. For any conflicting undo transaction pair U_i and U_j, if $T_i <_H T_j$ then $U_i <_H U_j$, and
5. For any undo operation $w_U[x]$, the predicate $(x \in \mathit{dirty\_item\_set}) \cap (x \in \mathit{submitted\_item\_set})$ holds.

Termination Detection

This is the same as Mechanism 2 except that we maintain the *tmp_undo_list* in the following way:

- For each in-repair transaction T, if $T.Read\_Set \cap dirty\_item\_set \neq \emptyset$, then put T into the list.
- For each in-repair transaction T, if $\exists x \in T.Read\_Set \cap cleaned\_item\_set$ such that $T.Begin.LSN \leq x.LSN$, then put T into the list.

It should be noticed that here the conditions which are used to detect termination are only adequate, but not necessary. That is, when the conditions are satisfied, the repair terminates; but when the repair terminates, these conditions may not be satisfied.

Building Undo Transactions

The same as Algorithm 4.

On-the-fly Repair Algorithm

Here, we specify only the modifications to Algorithm 6.

Algorithm 9 On-the-fly Repair Algorithm

```
At the Repair Manager
        The case in Algorithm 6 for a read record $[T_i, x]$ is removed
        case a commit record $[T_i, commit]$ is scanned
            if $T_i.Read\_Set \cap dirty\_item\_set \neq \emptyset$
                build the undo transaction for $T_i$ and submit it.
                put all items of $T_i$ into the $dirty\_item\_set$.
            if $\exists x \in T_i.Read\_Set \cap cleaned\_item\_set$ such that $T_i.Begin.LSN \leq x.LSN$
                build the undo transaction for $T_i$ and submit it.
                put all items of $T_i$ into the $dirty\_item\_set$.
```

5.4.6 Getting Write Items from the Log

Since each read set in a transaction profile is a set of logical identifiers for data items, we need to relate these items with the write set data items we get from the log. As described in [GR93], there are typically four ways to maintain tuple identifications in database systems, which are Relative

Byte Addresses(BRAs), Tuple Identifiers(TIDs), Database Keys and Primary Keys. The advantages of the primary key addressing technique outweigh its higher cost, especially since there are optimizations to reduce the overhead for many operations. Even if we can not get the logical identifiers of write set data items from the log directly, we can map from their physical identifiers to logical identifiers based on the internal mapping and addressing mechanisms of database systems.

5.5 Discussion

5.5.1 Comparison of the Performance of Different Repair Approaches

As shown in Section 5.2, Section 5.3 and Section 5.4, keeping read information in the log can achieve an exact repair, but it may incur substantial performance penalties due to the significant storage and processing cost of maintaining read information.

Extracting read sets from transaction profiles cuts the extra cost significantly, but it usually can only achieve a complete repair, and not an exact repair. That is, some non-suspect good transactions may have to be undone. In dynamic repair there may be a delay in detecting termination.

Maintaining a special purpose graph with transaction dependency information has many attractions: the graph is immediately available for backing out undesirable transactions, the frequency with which read information is put into stable storage can be dynamically adjusted as appropriate, and the graph can be targeted to cover those transactions most likely to be marked undesirable. For example, in the case of the upgrade problem, it is easy to identify the source of potential undesirable transactions, and the protection gained by being able to back out such transactions warrants a short term sacrifice in performance.

5.5.2 Incorporating Ongoing Attacks

In static repair, due to the delay of intrusion detection, a bad transaction may be identified during the repair; Similarly in dynamic repair, new bad transaction can be identified at any time during the repair. For simplicity of presentation, we assume that the malicious transaction list will not change in the process of both static and dynamic repairs. However, ongoing attacks can be incorporated into our algorithms.

In static repair, newly detected bad transactions can be repaired by re-scanning the whole log. In dynamic repair, when a new bad transaction is identified, we stop the repair, skip to the place where the first un-repaired bad or suspect transaction begins, and apply the dynamic repair algorithm again. In the new round, the new bad and suspect transactions are backed out.

SECTION 6

Trusted Recovery by Rewriting Histories

This chapter presents the semantic aspect of the proposed trusted recovery scheme where a set of rewriting algorithms is developed and substantial transaction semantics is incorporated to save the work of more good transactions.

6.1 The Model

We consider a history H over $\mathbf{B} \cup \mathbf{G}$. We assume that the concurrency control mechanism provides an explicit serial history H^s of history H. For example, the order of first lock release provides a serialization order for transactions scheduled by a strict two-phase locking mechanism. We denote the total order on the transactions in a serial history H^s by $<^s_H$.

We assume the availability of read information for transactions in H since as later discussion makes clear, read information is also necessary to rewrite histories. Read information can be captured in several ways, these approaches are discussed in section 6.6.

We assume that transactions do not issue blind writes. Although the approach in this chapter can be adapted to blind writes, doing so complicates the presentation. Also, we compare the results in this chapter to those obtained by a dependency-graph based approach to recovery (proposed in Chapter 5) that also assumes no blind writes.

Figure 6.1 illustrates the dependency-graph based approach to backing out bad and affected transactions (see Chapter 5). In particular, it illustrates the importance of distinguishing between read-write and write-read dependencies during recovery. A read-write edge can leave the 'zone of repair' without causing the zone to expand. On the other hand a write-read edge potentially expands the zone. Note that due to the assumption of no blind writes, there are no write-write edges in the graph.

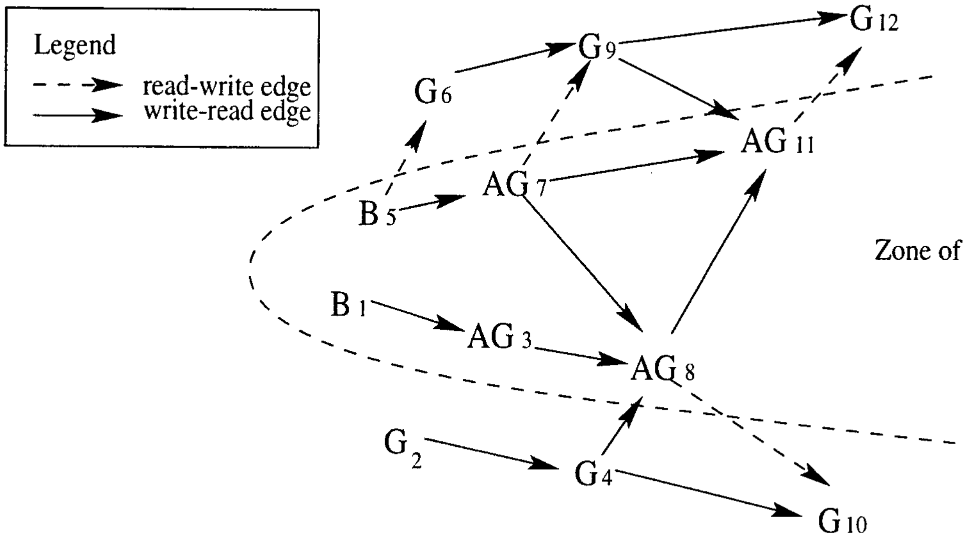


Figure 6.1: Zone of Repair

In this example, a possible history H_4 is

$$H_4 = B_1\ G_2\ AG_3\ G_4\ B_5\ G_6\ AG_7\ AG_8\ G_9\ G_{10}\ AG_{11}\ G_{12}$$

the set $\mathbf{AG} = \{AG_3,\ AG_7,\ AG_8,\ AG_{11}\}$, and the dependency-graph based recovery algorithms proposed in Chapter 5 restore the before values for all data items written by transactions in the set $\mathbf{B} \cup \mathbf{AG}$. The result is a serializable history over $\mathbf{G} - \mathbf{AG}$:

$$H_{4r} = G_2\ G_4\ G_6\ G_9\ G_{10}\ G_{12}$$

The approach of rewriting histories developed in this chapter has the advantage that it preserves ordering information for transactions in $\mathbf{B} \cup \mathbf{AG}$, thereby providing a basis for saving additional transactions in $\mathbf{AG}$.

6.1.1 Rewriting Histories

For a serial history H^s, we *augment* H^s with explicit database states so that the result is a sequence of interleaved transactions and database states. The sequence begins and ends with a state. The state that immediately precedes a transaction in H^s is called the *before state*; the state that immediately follows a transaction in H^s is called the *after state*. For an example, consider the augmented history

$H_5^s = s_0 \; B_1 \; s_1 \; G_2 \; s_2$

where

$B_1 : \texttt{if } x > 0 \texttt{ then } y := y + z + 3$
$G_2 : x := x - 1$

The states associated with H_5^s are:

$s_0 = \{x = 1; \;\; y = 7; \;\; z = 2\}$
$s_1 = \{x = 1; \;\; y = 12; \;\; z = 2\}$
$s_2 = \{x = 0; \;\; y = 12; \;\; z = 2\}$

In rewriting histories, the general goal is either to move bad transactions towards the end of a history or to move good transactions towards the beginning of a history. It turns out that the transformations do not necessarily result in a serializable history which is conflict-equivalent or view-equivalent to the original history[BHG87]. The lack of serializability is justified by the observation that bad transactions ultimately must be backed out anyway along with some or all of the affected transactions. Hence the serializability of such transactions is not a requirement.

The example above helps to clarify this point. The serial history H_5^s is clearly not conflict-equivalent to the serial history G_2B_1 since there is a read-write dependency from B_1 to G_2. However, G_2 is not affected by B_1, and simply restoring y with the appropriate value from the log not only repairs the damage caused by B_1, but preserves the effects of the good transaction G_2.

However, It turns out that rewriting histories for recovery purposes requires some care with respect to state-equivalence of histories. Two augmented histories H_1^s and H_2^s are *final state equivalent* if they are over the same set of transactions and the final states are identical. Note that two final state equivalent histories might not be conflict-equivalent, or view-equivalent [BHG87].

To clarify this point, consider the above example again. After we make the transformation of exchanging the order of G_2 and B_1, H_1^s is clearly not final state equivalent to the serial history G_2B_1 since they result in different final states. At this situation, if H_1^s has more transactions following B_1G_2, i.e., $G_3G_4...G_n$, then this transformation changes the before state of G_3. As a result, after the transformation the rewritten history may not be consistent any longer because the precondition of some G_i, $3 \leq i \leq n$, may not be satisfied any more. Even if the rewritten history is still consistent, the behaviors and effects of G_3, G_4, ..., and G_n may have changed a lot, thus the original execution log may turn out to be useless. Moreover, the rewritten history usually can not result in the same final state, and the new final state is usually very difficult to get, thus semantics-based compensation is disabled. Therefore, keeping the final state equivalence of rewritten histories during a rewrite is essential to the success of the rewrite.

We approach this problem by decorating each transaction T in an augmented history H^s with special values for read purposes by T. The decoration is facilitated by the notation *fix* which is specified below.

Definition 5 A *fix* for transaction T_i in history H^s, denoted F_i, is a set of variables read by T given values as in the original position of T in H^s. That is, $F_i = \{(x_1, v_1), ..., (x_n, v_n)\}$, and v_i is what T_i read for x_i in the original history.

The notation ${T_i}^{F_i}$ indicates that the values read by T_i for variables in F_i should not come from the before state of T_i, but from F_i.

To reduce notational clutter, we show just the variable names in F_i and omit the associated values.

Consider the augmented history $H_5^s = s_0\ B_1\ s_1\ G_2\ s_2$ above. As discussed, the history

$$H_6^s = s_0\ G_2\ s_3\ B_1\ s_3$$

with

$$s_3 = \{\mathrm{x} = 0;\ \ \mathrm{y} = 7;\ \ \mathrm{z} = 2\}$$

results in a different value of y in the final state, but the history

$$H_7^s = s_0\ G_2\ s_3\ B_1^{F_1}\ s_2$$

ends in final state s_2 for $F_1 = \{x\}$. States s_1 and s_3 differ in the value of x; this discrepancy is captured by F_1, where x is associated with the value 1, which is the value B_1 read for x in the original history H_5^s.

In what follows, each transaction T_i is assumed to have an associated fix F_i. For ordinary serializable execution histories, each such fix $F_i = \emptyset$, the empty fix. In the example above, the two histories

$$H_5^s = s_0\ B_1^{\emptyset}\ s_1\ G_2^{\emptyset}\ s_2$$
$$H_7^s = s_0\ G_2^{\emptyset}\ s_3\ B_1^{\{x\}}\ s_2$$

are final state equivalent.

6.1.2 Repaired Histories

Definition 6 Given a history H^s over $\mathbf{B} \cup \mathbf{G}$, H^s_r is a *repaired* history of H^s if

1. H^s_r is over some subset of $\mathbf{G}$, and
2. There exists some history H^s_e over $\mathbf{B} \cup \mathbf{G}$ such that
   (a) H^s_r is a prefix of H^s_e and
   (b) H^s_e and H^s are final state equivalent.

Our notion of a repaired history is that only good transactions remain (condition 1) and further that there is some extension to the repair that captures exactly the same transformation to the database state as the original history (condition 2).

We note that the dependency-graph based approach satisfies the first part of the definition of a repaired history where the subset of $\mathbf{G}$ is $\mathbf{G} - \mathbf{AG}$. As an example, in figure 6.1 history H^s_{4r} is a repair of H^s_4 since H^s_{4r} is over $\{G_2, G_4, G_6, G_9, G_{10}, G_{12}\}$ which is a subset of $\mathbf{G}$ and the necessary history H^s_{4e} exists:

$$H^s_{4e} = G_2\ G_4\ G_6\ G_9\ G_{10}\ G_{12}\ B_1^{F_1}\ AG_3^{F_3}\ B_5^{F_5}\ AG_7^{F_7}\ AG_8^{F_8}\ AG_{11}^{F_{11}}$$

for appropriate fixes F_1, F_3, F_5, F_7, F_8 and F_{11}. Details of how to construct fixes are discussed later in the chapter.

Armed with a definition of repairs to histories, we are now ready to consider algorithms to construct them.

6.2 Basic Algorithm to Rewrite a History

6.2.1 Can-Follow Relation

We denote the set of items read or written by a transaction T as *T.readset* or *T.writeset*, and the set of items read or written by a sequence of transactions $R = T_1T_2...T_n$ as *R.readset* or *R.writeset*. Due to our assumption of no blind writes, $R.writeset \subseteq R.readset$.

Definition 7 Transaction T *can follow* a sequence of transactions R if

$$T.writeset \cap R.readset = \emptyset$$

There are some properties of can follow:

1. If $T_i.writeset$ is not empty, then transaction T_i can not follow itself.
2. The fact that T_i can follow transaction T_j and T_j can follow transaction T_k does not imply that T_i can follow T_k.
3. Read-only transactions can follow any transaction.

The can follow relation captures the notion that a transaction T can be moved to the right past a sequence of transactions R if no transaction in R reads from T. The can follow relation ensures then the cumulative effects of the transactions in R on the database state are identical both before and after T is moved. The following lemma shows that the can follow relation can be repeatedly used to rewrite a history.

Lemma 2 Transaction T can follow a sequence of transactions R iff T can follow every transaction in R.

Proof: *if:* For every transaction T_i in R, $T.writeset \cap T_i.readset = \emptyset$ because T can follow T_i. Therefore $T.writeset \cap R.readset = \emptyset$, so T can follow R.

only if: By contradiction, assume there is a transaction T_i in R such that T cannot follow T_i, then $T.writeset \cap T_i.readset \neq \emptyset$. Therefore, $T.writeset \cap R.readset \neq \emptyset$, which contradicts the assumption that T can follow R. □

6.2.2 Can-Follow Rewriting

The can follow relation can be used to rewrite a history to move transactions in $\mathbf{G} - \mathbf{AG}$ to the beginning of the history, namely, move transactions in $\mathbf{B} \cup \mathbf{AG}$ backwards.

Algorithm 10 Can-Follow Rewriting
Input: the serial history H^s to be rewritten and the set $\mathbf{B}$ of bad transactions.
Output: a rewritten history with transactions in $\mathbf{G} - \mathbf{AG}$ preceding transactions in $\mathbf{B} \cup \mathbf{AG}$.
Method: Scan forward from the first good transaction after B_1 until the end of H^s, for each transaction T

```
    case $T \in \mathbf{B}$    skip it;
    case $T \in \mathbf{G}$
        if each transaction between $B_1$ and $T$ (including $B_1$) can follow $T$, then
        move $T$ to the position immediately preceding $B_1$.
```

Algorithm 10 does not describe how to compute the *fix* with any transaction which has some transaction being moved to the left of it. The reason is that repair can simply be accomplished by undo. However, if we want to save some of the transactions in **AG** then we need to maintain the *fix* information for these transactions. Fixes are computed as follows:

Lemma 3 Suppose transaction T can follow sequence R in history $H_1^s = s_0\ T^{F_1}\ s_1\ R\ s_2$. Then for fix

$$F_2 = F_1 \cup (T.readset \cap R.writeset)$$

history $H_2^s = s_0\ R\ s_3\ T^{F_2}\ s_2$ is final state equivalent to H_1^s. The values associated with each data item in the fixes are those originally read by T.

Proof: Consider some database item $x \in s_2$. x is not an element of both *R.writeset* and *T.writeset* since otherwise the relation T can follow R would not hold. If x is an element of *R.writeset*, then the value computed by R for x is the same in both H_1^s and H_2^s since R does not read from T. If x is an element of *T.writeset*, then the value computed by T for x is the same in both H_1^s and H_2^s since T reads identical values for elements in *T.readset* in both histories, courtesy of fixes F_1 and F_2, respectively. If x is not an element of either *T.writeset* or *R.writeset*, then the order of T and R is irrelevant to the value of x. □

The correctness of Algorithm 10 is specified as follows.

Theorem 7 Given a history H^s, Algorithm 10 produces a history H_e^s with a prefix H_r^s such that:

1. All and only transactions in **G** − **AG** appear in H_r^s.

2. H_e^s and H^s order transactions in **G** − **AG** identically. And they order transactions in **B** ∪ **AG** identically.

3. The fix associated with each transaction in H_r^s is empty.

4. H^s and H_e^s are final state equivalent. And H_r^s is a repaired history of H^s.

Proof: (1) We first show that when a transaction $T_1 \in$ **G** − **AG** is scanned every transaction between B_1 and T_1 is in **B** ∪ **AG**. Assume this is not the situation and T_2 is the first one between B_1 and T_1 which belongs to **G** − **AG**. According to the algorithm when T_2 was scanned it should be moved to the left of B_1, which is a contradiction. We second show that no transactions in **AG** will be moved to the left of B_1 at the end of the algorithm. Assume this is not the situation and T_2 is the first one in **AG**. According to the definition of **AG**, when T_2 is scanned there is at least

one transaction between B_1 and T_2 which can not follow T_2, which is a contradiction. We last show that no transactions in $\mathbf{B}$ will be moved to the left of B_1 because they will never be moved at all. Therefore, after the rewrite all and only transactions in $\mathbf{G} - \mathbf{AG}$ are moved to the left of B_1.
(2) Since Algorithm 10 moves transactions in $\mathbf{G} - \mathbf{AG}$ to the left of B_1 according to their orders in H^s, so they are ordered by H^s_r and H^s identically. Since transactions in $\mathbf{B} \cup \mathbf{AG}$ are never moved in Algorithm 10, so they are ordered by H^s_r and H^s identically.
(3) Since there are no transactions which are moved to the left of any transaction in $\mathbf{G} - \mathbf{AG}$ in Algorithm 10, transactions in $\mathbf{G} - \mathbf{AG}$ will have empty fixes.
(4) Follows from Lemma 3 and Definition 6. □

In realistic applications, although Lemma 3 gives users a sound approach to capture fixes in Algorithm 10, it is not efficient in many cases since whenever a transaction T_i is moved to the left of another transaction T_j, F_j may need be augmented. A better way to compute fixes is as follows:

Lemma 4 For any history H^s, assume rewriting H^s using Algorithm 10 generates a history H^s_e with a prefix H^s_r (H^s_e typically looks like:
$G_{j1}...G_{jn}\ B_{i1}^{F_{i1}}\ AG_{k1}^{F_{k1}}...B_{im}^{F_{im}}...AG_{kp}^{F_{kp}}$. The subhistory before $B_{i1}^{F_{i1}}$ is H^s_r), and assume all the fixes are computed according to Lemma 3 during the rewriting, then the history $H^{s\prime}_e$, generated by replacing each non-empty fix F_i in H^s_e with $F'_i = T_i.readset - T_i.writeset$, is final state equivalent to H^s_e.

Proof: According to Theorem 7, the fix associated with each transaction in H^s_r is empty. Given a transaction T_i in $\mathbf{B} \cup \mathbf{AG}$, for each item x in $F'_i - F_i$, showing that the value of x in the before state of T_i in H^s_e is the same as that in H^s gives the proof. Assume G_j is the first transaction which was moved to the left of T_i, then before G_j was moved, the before state of T_i in the rewritten history is the same as that in H^s because at this point, according to Lemma 3, the subhistory of the rewritten history which ends with the transaction immediately preceding T_i is final state equivalent to the corresponding subhistory of H^s. After G_j is moved to the left of T_i, the value of x would not be changed since otherwise x must be in F_i. Although G_j might be further pushed through some other transactions in $\mathbf{B} \cup \mathbf{AG}$ to the beginning of the history, the value of x in the before state of T_i will not be changed. The reason follows from Lemma 3. □

Lemma 4 enables us to separate computing fixed from transforming histories. Fixes can be computed after all of the transformations. Based on Lemma 4, the fix of transaction T_i can be captured in two ways: one is to first get the read and write sets of T_i, then compute $T_i.readset - T_i.writeset$; the other is to let each transaction T_i write the set $T_i.readset - T_i.writeset$ as a record to the database when it is executed, then when we rewrite H^s all of the fixes can be gotten directly from the database.

6.2.3 Significance of Algorithm 10

The major result of this section is an equivalence theorem between the effect of a dependency-graph based algorithm and the history produced by Algorithm 10. The dependency-graph based algorithm computes the set $\mathbf{B}.\mathit{writeset} \cup \mathbf{AG}.\mathit{writeset}$ and restores the values of all elements in this set. In particular, the theorem shows that the optimizations in the following section are strict improvements over the dependency-graph based algorithm.

Theorem 8 Given H^s, let H^s_r be the serial history produced by eliminating all transactions in $\mathbf{B} \cup \mathbf{AG}$ as in the dependency-graph based algorithm. Given H^s, let H^s_e be the result of Algorithm 10. Then H^s_r is a prefix of H^s_e.

Proof: Direct corollary of Theorem 7. □

6.3 Saving Additional Good Transactions

In this section, we show how to integrate the notion of commutativity with Algorithm 10 to save not only the transactions in $\mathbf{G} - \mathbf{AG}$, but potentially transactions in $\mathbf{AG}$ as well.

6.3.1 Motivating Example

Consider the following history:

$H_8 : B_1G_2G_3$
B_1: if $u > 10$ then $x := x + 100, y := y - 20$
G_2: $u := u - 20$
G_3: $x := x + 10, z := z + 30$

According to Algorithm 10, which rewrites based on can follow, G_3 needs to be undone since it reads from B_1 and hence is an element of $\mathbf{AG}$. The result of Algorithm 10 is the history $H^s_e = G_2B_1^{\{u\}}G_3$. Note that G_3 *commutes backward through* $B_1^{\{u\}}$ for any value of u*, and so an final state equivalent history is $G_2G_3B_1^{\{u\}}$. Compensation for $B_1^{\{u\}}$ can be applied directly to this history, but an undo approach requires more care. Suppose we decide to undo B_1 by restoring the before values for x

*We adapt the notation of commutativity from [LMWF94, Wei88]. Transaction T_2 *commutes backward through* transaction T_1 if for any state s on which T_1T_2 is defined, $T_2(T_1(s)) = T_1(T_2(s))$; T_1 and T_2 *commute* if each commutes backward through the other. Note that one-sided commutativity (i.e., commutes backward through) is enough for our purpose.

and y from the log entries for B. After B is undone the value of u is unchanged because only G_1 updates u. The value of z is unchanged because only G_3 updates z. The effect of G_3 on x is wiped out because both G_3 and B update x, and after B is undone x no longer reflects the effects of G_3. However x can be repaired by re-executing the corresponding part of G_3's code, that is, $x = x + 10$, and the cumulative effect is that of history G_2G_3. We call this last step an *undo-repair* action. Both the undo approach and the compensation approach to repair are discussed in detail in section 6.4.

The presence of fixes for transactions limits the extent to which commutativity can be applied. We illustrate this point with an example, and then define a more restrictive notion of commutativity called *can precede* that takes fixes into account.

$H_9 :\ s_0\ T_1\ s_1\ T_2\ s_2\ T_3\ s_3$
T_1: if $y > 200$ then $x := x + 100$ else $x := x * 2$
T_2: $y := y + 100$
T_3: if $y > 200$ then $x := x - 10$ else $x := x/2$

T_1 can follow T_2 with fix $F_1 = \{y\}$ for T_1. Although T_3 commutes backward through T_1, T_3 does not commute backward through $T_1^{F_1}$, because the value of x produced by $T_1^{F_1}$ depends on the value of y in the fix F_1. For example, if the initial value of x is 100 and fix value of y is 150, then the final value of x in history $T_2T_1^{F_1}T_3$ is 190, but the final value of x in history $T_2T_3T_1^{F_1}$ is 180.

The example shows that sometimes a fix can interfere with the commutativity of transactions. This motivates our definition of *can precede*:

Definition 8 A transaction T_2 *can precede* a transaction T_1 for fix F if for any assignment of values to the variables in F and for any state $s_0 \in \mathbf{S}$ on which $T_1^FT_2$ is defined,

1. $T_2T_1^F$ is defined on s_0, and

2. The same final state is produced by $T_1^FT_2$ and $T_2T_1^F$.

6.3.2 Can-Follow and Can-Precede Rewriting

We present a repair algorithm which integrates both can-follow and can-precede.

Algorithm 11 Can-Follow and Can-Precede Rewriting
Input: the history H^s to be repaired.
Output: the repaired history H_r^s.
Method: Scan H^s forward from the first good transaction after B_1 until the end of H^s, for each

```
transaction $T$
    case $T \in \mathbf{B}$    skip it;
    case $T \in \mathbf{G}$
        if for each transaction $T'$ between $B_1$ and $T$(including $B_1$), either $T'$ can follow $T$ or $T$ can precede $T'$, then move $T$ to the position immediately preceding $B_1$. As $T$ is pushed through each such $T'$ between $B_1$ and $T$ to the left of $B_1$
            if $T'$ can follow $T$, then push $T$ to the left of $T'$ and
             modulate the fix of $T'$ correspondingly according to Lemma 3;
            else push $T$ to the left of $T'$.
```

The correctness of Algorithm 11 is specified as follows.

Theorem 9 Given a history H^s, Algorithm 11 produces a history H_e^s with a prefix H_r^s such that:

1. Every transaction in $\mathbf{G} - \mathbf{AG}$ appears in H_r^s.

2. H_e^s and H^s order transactions in H_r^s identically. And they order transactions in $H_e^s - H_r^s$ identically.

3. The fix associated with each transaction in H_r^s is empty.

4. H^s and H_e^s are final state equivalent. And H_r^s is a repaired history of H^s.

Proof: The proof of statements (1), (2), and (3) is similar to that of Theorem 7.
(4) follows from Lemma 3, Definition 8 and Definition 6. □

In Algorithm 10, Lemma 4 provides an efficient way to compute fixes. However, Lemma 4 may not hold for Algorithm 11 if the system does not have the following property.

Property 1 Transaction T_j can precede transaction T_i for a fix F_i only if $(T_i.readset - T_i.writeset - F_i) \cap T_j.writeset = \emptyset$ and $(T_j.readset - T_j.writeset) \cap T_i.writeset = \emptyset$.

It should be noticed that Property 1 is not a strict requirement, and it usually holds for most of the transaction processing systems. The reason is: if T_j writes an item x in $T_i.readset - T_i.writeset - F_i$, then x can have different values in the before states of T_i in sequences $T_i^{F_i} T_j$ and $T_j T_i^{F_i}$ respectively. Since x is not in F_i, T_i can read different values of x in the two sequences. Since the value of x typically affects the values of some other items updated by T_i, the two sequences usually can not generate the same final state. For similar reasons, if $(T_j.readset - T_j.writeset) \cap T_i.writeset \neq \emptyset$, then $T_i^{F_i} T_j$ and $T_j T_i^{F_i}$ usually can not generate the same final state.

Lemma 5 Lemma 4 holds for Algorithm 11 if the system has Property 1.

Proof: The proof is similar to that of Lemma 4 except the situation when T_j is moved to the left of T_i based on the relation that T_j can precede T_i. At this point, for each item x in $F_i' - F_i$, since the system has Property 1, T_j will not write x, so the value of x in F_i' is still the same as that in the before state of T_i after the rewrite. This completes the proof. □

6.3.3 Invert and Cover

In this section, we introduce two semantic relationships between transactions, namely, *Invert* and *Cover*, and show how they can be exploited to enhance repair.

If transaction T_2 *inverts* T_1, then any history of the form: $s_0 ... T_1\ T_2\ ...$ is final state equivalent to the same history with $T_1 T_2$ omitted; if T_2 *covers* T_1, then any history of the form: $s_0 ... T_1\ T_2\ ...$ is final state equivalent to the same history with T_1 omitted. If T_2 covers T_1, then T_2 covers $T_1^{F_1}$ for any F_1, but this is not the case for invert.

Definition 9 Let P and Q be two sequences of transactions. Q *inverts* P if for any state s_0 such that history $s_0\ P\ Q$ is feasible, $Q(P(s_0)) = s_0$.

Definition 10 Let P and Q be two sequences of transactions. Q *covers* P if for any state s_0 such that history $s_0\ P\ Q$ is feasible, $Q(P(s_0)) = Q(s_0)$.

The rewriting algorithm which exploits these two relations is described below.

Algorithm 12 Can-Follow, Can-Precede, Cover, and Invert Rewriting
Input: the history H^s to be repaired.
Output: the repaired history H_r^s.
Method: Scan H^s forward from the first good transaction after B_1 until the end of H^s, for each transaction T

```
    case $T \in \mathbf{B}$    skip it;
    case $T \in \mathbf{G}$
        if for each transaction $T'$ between $B_1$ and $T$(including $B_1$), either $T'$
         can follow $T$, or $T$ can precede $T'$, or $T$ inverts $T'$, or $T$ covers $T'$,
         then move $T$ to the position immediately preceding $B_1$. As $T$ is pushed
         through each $T'$ between $B_1$ and $T$ to the left of $B_1$
            if $T$ covers $T'$, then remove $T'$ from the history;
            elseif $T'$ can follow $T$, then push $T$ to the left of $T'$ and
             modulate the fix of $T'$ correspondingly according to Lemma 3;
    elseif $T$ can precede $T'$, then push $T$ to the left of $T'$;
    else remove both $T$ and $T'$ from the history.
```

For similar reasons, Lemma 4 can also be exploited to capture fixes in Algorithm 12 if the system has Property 1. The correctness of Algorithm 12 is specified as follows. The proof is similar to Theorem 7 and Theorem 9, thus omitted.

Theorem 10 Given a history H^s, Algorithm 12 produces a history H_e^s with a prefix H_r^s such that:

1. H_e^s and H^s order transactions in H_r^s identically. And they order transactions in $H_e^s - H_r^s$ identically.
2. The fix associated with each transaction in H_r^s is empty.
3. Every transaction in H_e^s is in H^s.
4. The final states of H^s and H_e^s are identical. And H_r^s is a repaired history of H^s.

6.4 Pruning Rewritten Histories

After a rewritten history H_e^s with a prefix H_r^s , which is the repaired history, is generated from H^s, we need to prune H_e^s such that the effects of all the transactions in $H_e^s - H_r^s$ are removed. Pruning H_e^s generates H_r^s. If H_e^s is produced by Algorithm 10, then the pruning can be easily done by undoing each transaction in $H_e^s - H_r^s$. However, if H_e^s is produced by Algorithm 11 or Algorithm 12, undo does not give the pruning in most cases.

In this section, two pruning approaches are presented. The compensation approach removes the effect of each transaction $T_i^{F_i}$ in $H_e^s - H_r^s$ by executing the *fixed* compensating transaction of T_i, however, compensating transactions may not be specified in some systems. The undo approach prunes H_e^s by building and executing a specific undo-repair action for each affected transaction in H_r^s. It is a syntactic approach, but it imposes some restrictions on transaction programs.

6.4.1 The Compensation Approach

We denote the compensating transaction of transaction T_i as T_i^{-1} [GM83, GMS87, KLS90]. T_i^{-1} semantically undoes the effect of T_i. It is reasonable to assume that $T_i^{-1}.\mathit{writeset} \subseteq T_i.\mathit{writeset}$, and for simplicity we further assume that every transaction T_i has a compensating transaction.

After Algorithm 11 or Algorithm 12, a typical rewritten history H_e^s with a prefix H_r^s looks like (note that B_{i1} could be covered or inverted, and H_e^s can also end with a bad one): $G_{j1}...AG_{h1}...G_{jq}...AG_{hk}\ B_i$ The subhistory before $B_{i1}^{F_{i1}}$ is H_r^s. Based on H_e^s, compensation is a simple way to get the repaired history H_r^s. However, executing the compensating transaction sequence $AG_{hp}^{-1}...B_{im}^{-1}...AG_{h(k+1)}^{-1}\ B_{i1}^{-1}$ on the final state of H^s can not generates H_r^s in most cases because the transactions we need to compensate are usually associated with a non-empty fix. Fixes must be taken into account for the compensation to be correct.

Definition 11 The *fixed compensating transaction* of $T_i^{F_i}$, denoted $T_i^{(-1,F_i)}$, is the regular compensating transaction of T_i (denoted T_i^{-1}) associated with the same fix F_i.

The effects of $T_i^{F_i}$ can be removed by executing $T_i^{(-1,F_i)}$, this is justified by the following lemma.

Lemma 6 Transaction $T_i^{F_i}$ can be *fix compensated*, that is, for every consistent state s_1 on which $T_i^{F_i}$ is defined, $T_i^{(-1,F_i)}(T_i^{F_i}(s_1)) = s_1$, if $F_i \cap T_i.writeset = \emptyset$.

Proof: Since $F_i \cap T_i.writeset = \emptyset$, $T_i^{-1}.writeset \subseteq T_i.writeset$, so $F_i \cap T_i^{-1}.writeset = \emptyset$. Therefore, neither T_i nor T_i^{-1} will update any item in F_i. Let $s_2 = T_i^{F_i}(s_1)$. For each item x in F_i we replace the values of x in states s_1 and s_2 with the value of x in F, thus two new states are generated (denoted s_1' and s_2' respectively). It is clear that $T_i^{-1}(s_2') = s_1'$. Since the differences between $T_i^{-1}(s_2')$ and $T_i^{(-1,F_i)}(s_2)$ are only with the values of the items in F_i which are neither updated by T_i^{-1}, nor updated by $T_i^{(-1,F_i)}$, so $T_i^{(-1,F_i)}(s_2) = s_1$. This completes the proof. □

A rewritten history H_e^s can be *fix compensated* if every transaction in H_e^s can be fix compensated. Lemma 6 shows that every H_e^s produced by Algorithm 11 or Algorithm 12 can be fix compensated because for each transaction T_i in H_e^s which is associated with a non-empty fix F_i, $F_i \cap T_i.writeset = \emptyset$ always holds. The pruning algorithm by compensation therefore is straightforward: based on the final state of H^s, executing the fixed compensating transaction for each transaction in $H_e^s - H_r^s$ in the reverse order as they are in H^s.

6.4.2 The Undo Approach

As stated above, after Algorithm 11 or Algorithm 12, a typical rewritten history H_e^s looks like: $G_{j1}...AG_{h1}...G_{jq}...AG_{hk}\ B_{i1}^{F_{i1}}\ AG_{h(k+1)}^{F_{h(k+1)}}...B_{im}^{F_{im}}...AG_{hp}^{F_{hp}}$. As shown in H_8, undoing transactions in $H_e^s - H_r^s$ can not generate H_r^s in most cases. However, building and executing the undo-repair actions for the affected transactions in H_r^s, namely $AG_{h1}, ..., AG_{hk}$, after these undo operations can generate H_r^s. For example, in H_8, executing the undo-repair action, $x = x + 10$, for G_3 after B is undone can produce the effect of history G_2G_3.

To build the undo-repair actions for $AG_{h1}, ..., AG_{hk}$, we need to do two things:

1. Abstract the code for each undo-repair action from the source code of the corresponding affected transaction.
2. Assign appropriate values for some specific data items accessed by these undo-repair actions.

Our algorithm described below is based on the following assumptions about transactions:

- a transaction is composed of a sequence of statements, each of which is either:
  - An operation;
  - A conditional statement of the form: **if** c **then** $SS1$ **else** $SS2$, where $SS1$ and $SS2$ are sequences of statements, and c is a predicate;
- each statement can update at most one data item;
- each data item is updated only once in a transaction;

```
Algorithm 13 Build Undo-repair Actions
Input: an affected transaction $AG_k$.
Output: the undo-repair action $URA_k$ for $AG_k$.
Method:
1. Copy the codes of $AG_k$ to $URA_k$. Assign $URA_k$ with the same input parameters and the same
values associated with them as $AG_k$.
2. Parse $URA_k$. For each statement to be scanned
    case it is a read statement, keep it;
    case it is an update statement of the form: $x := f(x, y_1, y_2, ...y_n)$ where $f$
     specifies the function of the statement, $y_1, ..., y_n$ are the data items used
     in the statement. Some input parameters may also be used in the statement,
     but they are not explicitly stated here.
      if $x$ has not been updated by any other transaction in $\mathbf{B} \cup \mathbf{AG}$
          Remove the statement from $URA_k$;
      elseif $x$ has not been updated by any transaction in $\mathbf{B} \cup \mathbf{AG}$
       which precedes $AG_k$ in $H^s$
          Replace the statement with: $x := AG_k.afterstate.x$,
          that is, get the value of $x$ from the after state of $AG_k$ in $H^s$;
```

```
else for each $y_i$ (including $x$)
  if $y_i$ has not been updated by any preceding statement and has not been
  updated by any transaction in $\mathbf{B} \cup \mathbf{AG}$ which precedes $AG_k$ in $H^s$
    Bind $y_i$ with $AG_k.beforestate.y$;
```

3. Reparse URA_k. Remove every read statement which reads some item never used in an update statement of URA_k, or reads some item y used in one or more update statements but y is bound with a value in these statements.

It should be noticed that when we execute an undo-repair action URA_k, for each update statement $x = f(x, y_1, y_2, ...y_n)$ of URA_k, if y_i is not bound then we get the value of y_i from the current database state, otherwise, the bound value should be used.

The correctness of the undo approach is specified as follows.

Theorem 11 For any rewritten history H_e^s generated by Algorithm 11 or Algorithm 12, after all transactions in $H_e^s - H_r^s$ are undone, executing the undo-repair actions which are generated by Algorithm 13 for the affected transactions in H_r^s, in the same order as their corresponding affected transactions are in H_r^s, produces the same effect of H_r^s.

Proof: Showing that each item x updated by an transaction in H_e^s is restored to the value as generated by H_r^s after the repair gives the proof.

If x has never been updated by any transaction in $\mathbf{B} \cup \mathbf{AG}$, then the value of x will be correctly restored because an unaffected transaction G_i can only read items from other unaffected transactions thus G_i's updates will not be affected by transactions in $\mathbf{B} \cup \mathbf{AG}$.

Otherwise, assume x has been updated by k transactions in $\mathbf{B} \cup \mathbf{AG}$, that is, $T_{i1}, ..., T_{ik}$, $T_{ip} <_H^s T_{iq}$ if $p < q$. Note that after x has been updated by T_{i1}, x will not then be updated by any unaffected transaction. If $k = 1$, that is, there is only one such transaction. At this point, if T_{i1} is in $H_e^s - H_r^s$ then after the undoes the value of x will be correctly restored; otherwise, T_{i1} is in H_r^s. Since H_e^s is final state equivalent to H^s, so the value of x in the final state of H_r^s is the same as that in the after state of T_{i1} in H^s. Hence in Algorithm 13 the corresponding update statement is removed.

When $k > 1$, if no such transaction is in H_r^s then after the undoes the value of x will be correctly restored. Otherwise, assume T_{j1} is in H_r^s, then when URA_{j1} is executed, $x := T_{j1}.afterstate.x$, according to Algorithm 13. This restores the value of x to that generated by the subhistory of H_r^s which ends with T_{j1}, because in rewriting when T_{j1} is moved into H_r^s, the subhistory H_1 of H^s which ends with T_{j1} is final state equivalent to the subhistory H_2 of the rewritten history at that time which ends with the transaction immediately preceding T_{j1} before the move, and T_{j1} is the last transaction in H_2 that updates x.

Assume T_{jl} ($l > 1$) is in H_r^s, if there is another such transaction T_{jm} in H_r^s such that $1 \leq m < l$ and no other such transactions stay between T_{jm} and T_{jl}, then in the update statement

$x := f(x, y_1, y_2, ...y_n)$ of URA_{jl}, the value of x for read purpose should be got from the state after URA_{jm} is executed; otherwise, there is no such T_{jm}, thus transactions $T_{j1}, ..., T_{j(l-1)}$ will all be undone, hence the value of x in the above statement should be got from the state after T_{j1} is undone.

As for y_i in the above update statement, if y_i has been updated by a preceding statement in URA_{jl}, then the updated value should be used. Otherwise, if y_i has been updated by some transaction in $\mathbf{B} \cup \mathbf{AG}$ which precedes T_{jl} in H^s, then according to the above discussion, the value of y_i should be got from the state before URA_{jl} is executed; Otherwise, the value of y_i should be got from the before state of T_{jl} in H^s. At this situation, getting the value of y_i from the state before URA_{jl} is executed can not ensure the correctness because it is possible that there is a transaction T_i such that T_i updates y_i, T_i follows T_{jl} in H^s, T_i is in $\mathbf{B} \cup \mathbf{AG}$ and T_i is in H^s_r. At this point, the value of y_i updated by T_i will not be undone.

Since the values of $x, y_1, y_2, ...y_n$ in the above statement are correctly captured, so the above statement can correctly restore the value of x to that generated by the subhistory of H^s_r which ends with T_{jl}. By induction on l, $1 \leq l \leq k$, the above claim holds. □

6.5 Relationships between Rewriting Algorithms

Rewriting can save more good transactions than is possible with a dependency-graph based approach to recovery. For a history H^s to be repaired, we will let DGR(H^s) and CFR(H^s) represent the sets of saved transactions after H^s is repaired using a dependency-graph based approach and can-follow rewriting (Algorithm 10), respectively. FPR(H^s) and FPCI(H^s) will be used to represent the sets of saved transactions after H^s is repaired using can-follow and can-precede rewriting (Algorithm 11) and can-follow, can-precede, cover and invert rewriting (Algorithm 12), respectively.

Theorem 8 shows that for any history H^s, DGR(H^s) = CFR(H^s).

Theorem 12 For any history H^s, CFR(H^s) $\subseteq$ FPR(H^s) $\subseteq$ FPCI(H^s). The converse is not, generally, true.

Proof: Follows from Algorithm 10, Algorithm 11, and Algorithm 12. □

Commutativity can be directly used to rewrite histories without being integrated with can-follow rewriting. Let CR(H^s) and CBTR(H^s) represent the sets of saved transactions after H^s is repaired using the two rewriting algorithms which are based on the *commute* relation and the *commutes backward through* relation between transactions, respectively. These two algorithms can be easily adapted from Algorithm 10 by checking the *commute* and *commutes-backward-through* relation between transactions respectively, instead of *can-follow*.

Theorem 13 For any history H^s, CR(H^s) $\subseteq$ CBTR(H^s). The converse is not, generally, true.

Proof: Follows from the definitions of *commute* and *commutes backward through.* □

Theorem 14 $\exists\ H^s$, CFR(H^s) $\cap$ CBTR(H^s) $\neq \emptyset$ *and* each is not included in the other; $\exists\ H^s$, CFR(H^s) $\cap$ CR(H^s) $\neq \emptyset$ *and* each is not included in the other;

Proof: Consider the history

$H_{10}:\ s_0\ B_1\ s_1\ G_2\ s_2\ G_3\ s_3$
B_1: if $y > 200$ then $x := x + 10$
G_2: if $y > 200$ then $x := x + 30$
G_3: $y := y + 100$

It is clear that CFR(H_{10}^s)= $\{G_3\}$; CBTR(H_{10}^s)=CR(H_{10}^s)= $\{G_2\}$. This completes the proof. □

Theorem 15 If the system has Property 1, then

1. $\forall\ H^s$, CBTR(H^s) $\subseteq$ FPR(H^s)
2. $\exists\ H^s$, CBTR(H^s) $\subset$ FPR(H^s)

Proof: Given a history H^s, showing that $T_i \in$ FPR(H^s) holds for each transaction $T_i \in$ CBTR(H^s) gives the proof. We prove this by induction on k where T_k is the kst transaction moved into CBTR(H^s).

Induction base: ($k = 1$) We wants to show that $T_1 \in$ FPR(H^s). If there are no transactions between B_1 and T_1 which are in FPR(H^s), then T_1 will be moved into FPR(H^s) according to Algorithm 11 because T_1 can precede every transaction $T_j^{\emptyset}$ between B_1 and T_1 owing to the fact that T_1 commutes backward through T_j. Otherwise, there must be some transaction T_j with an non-empty fix F_j staying between B_1 and T_1 (including B_1) in the rewritten history when T_1 is scanned in Algorithm 11. Here we assume that F_j is captured by Lemma 3. At this point, assume T_1 cannot precede $T_j^{F_j}$, then $F_j \cap (T_1.readset - T_1.writeset) \neq \emptyset$ because otherwise T_1 can precede $T_j^{F_j}$ (The reason is: for every state s_0 on which $T_j^{F_j}T_1$ is defined, replacing s_0 with another state s_1 where the value of each item x in $s_0 \cap F_j$ is replaced with x's value in F_j. Then $T_j^{\emptyset}T_i$ is defined on s_1. According to Property 1, since T_1 can precede $T_j^{\emptyset}$ (Note that T_1 commutes backward through T_j), so $(T_j.readset - T_j.writeset) \cap T_1.writeset = \emptyset$. Since $F_j \subseteq (T_j.readset - T_j.writeset)$ according to Lemma 3, so $F_j \cap T_1.writeset = \emptyset$. So T_1 will not read or update any item in F_j. Therefore, $T_j^{F_j}T_1(s_0) = T_j^{\emptyset}T_1(s_1)$, and $T_1T_j^{F_j}(s_0) = T_1T_j^{\emptyset}(s_1)$. Since T_1 commutes backward through T_j, so $T_j^{\emptyset}T_1(s_1) = T_1T_j^{\emptyset}(s_1)$. Therefore, $T_j^{F_j}T_1(s_0) = T_1T_j^{F_j}(s_0)$, so T_1 can precede $T_j^{F_j}$). Therefore, $\exists x$, such that, $x \in F_j \cap (T_1.readset - T_1.writeset)$. Since $x \in F_j$, so according to Algorithm 11 there must be a transaction T_p, such that T_p is now in FPR(H^s), and $x \in T_p.writeset$. Otherwise, x will not be put into F_j by Lemma 3. Hence $T_p.writeset \cap (T_1.readset - T_1.writeset) \neq \emptyset$. This conflicts with Property 1 since T_1 commutes backward through T_p thus T_1 can precede $T_p^{\emptyset}$. So the assumption that T_1 cannot precede $T_j^{F_j}$ does not hold. Therefore, T_1 can precede $T_j^{F_j}$. So T_1 can precede every transaction between B_1 and T_1 which has an non-empty fix. Since T_1 commutes backward through all the other transactions between B_1 and T_1, so T_1 will be moved into FPR(H^s).

Induction hypothesis: for each $1 \leq k \leq n$, if $T_k \in$ CBTR(H^s), then $T_k \in$ FPR(H^s).

Induction Step: Let $k = n + 1$, then when T_k is scanned in both algorithms, every transaction T_j, which is between B_1 and T_k in the rewritten history generated by Algorithm 11 at that time, is between B_1 and T_k in the rewritten history generated by the commutes-backward-through rewriting algorithm. Therefore, T_k commutes backward through every such T_j. For the same reason as in the *induction base* step, we know that T_k will be moved into FPR(H^s).

Therefore, statement 1 holds. Consider history H_{10}, it is clear that FPR(H_{10}^s)= $\{G_2, G_3\}$; CBTR(H_{10}^s) = $\{G_2\}$. So statement 2 holds. □

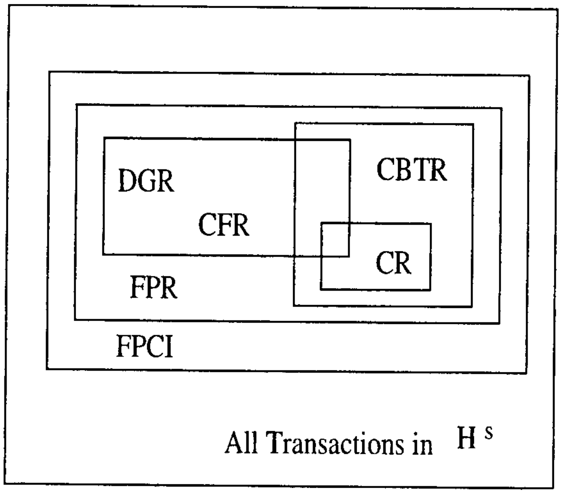
In summary, after a history H^s is repaired, the relationships among DGR(H^s), CFR(H^s), FPR(H^s), FPCI(H^s), CR(H^s) and CBTR(H^s) are shown in Figure 6.2. Here we assume that the system has Property 1.

6.6 Implementing the Repair Model on Top of Sagas

In this section, we will evaluate the feasibility of our repair model by integrating it with the saga model [GMS87].

6.6.1 The Saga Model

The Saga Model is a practical transaction processing model addressing long duration transactions which can be implemented on top of an existing DBMS without modifying the DBMS internals at all. A saga consists of a collection of saga transactions (or *steps*), each of which maintains database consistency. However any partial execution of the saga is undesirable; either all the transactions in a saga complete successfully or compensating transactions should be run to amend for the partial execution of the saga. Thus corresponding to every transaction in the saga, except the last one, a



DGR--set of transactions saved by a dependency-graph based approach

CFR--set of transactions saved by can-follow rewriting

CR--set of transactions saved by commute rewriting

CBTR--set of transactions saved by commutes-backward-through rewriting

FPR--set of transactions saved by can-follow and can-precede rewriting

FPCI--set of transactions saved by can-follow, can-precede, cover and invert rewriting

Figure 6.2: Relationships among Repair Approaches

compensating transaction is specified. The compensating transaction semantically undoes the effect of the corresponding transaction.

The Saga Model is suitable for our repair model to be implemented on top of it because it supports compensation inherently. For example, a compensating transaction is specified for each transaction, except the last one, in a saga; and when a saga transaction T_{ij} ends, the *end-transaction call* will include the identification of the compensating transaction of T_{ij} which includes the name and entry point of the compensating program, plus any parameters that the compensating transaction may need.

By viewing each normal duration transaction and each long duration transaction which can not be specified as a multi-step saga, as a specific saga that consists of only one saga transaction, we can get an unified view of transactions in the systems where the saga model is implemented. By adding the compensating transaction for the last step in each saga, we can get all the necessary compensating transactions to do repair.

In addition, the saga model has the following two features which allow for optimization in rewriting a history.

Consistency Property : the execution of each saga transaction (step) maintains database consistency.

Compensation Property : during the lifetime of a saga [†], no matter how the saga is interleaved with other sagas, any step in the saga which is successfully executed, if having not been compensated, can be compensated by executing the corresponding compensating transaction at the end of the growing history.

6.6.2 Repair a History of Sagas

The Compensation Property implies that in a history to be repaired whenever a saga is identified as a bad one, we can rewrite the history to move only the last step, instead of every step, of the saga to the end of the history. In this way, substantial rewriting and pruning work can be saved. The optimization based on can-follow rewriting (Algorithm 10) is specified in the the following algorithm.

Algorithm 14 Rewrite a history of sagas by can-follow rewriting
Input: the serial history H^s to be rewritten and the set **B** of bad sagas.
Output: a rewritten history H_e^s with a prefix H_r^s which consists of only good saga transactions.
Method: Scan forward from the first good saga transaction after B_{11} until the end of H^s, for each step T_{ij} (of saga S_i)

```
    case $S_i \in \mathbf{B}$    skip it;
    case $S_i \in \mathbf{G}$
        if there is a step of $S_i$ which stays between $B_1$ and $T_{ij}$
            Skip $T_{ij}$;
        elseif the final step $T_{pn}$ of every saga $S_p$ which stays between
        $B_1$ and $T_{ij}$ (including $B_1$) can follow $T_{ij}$
            Move $T_{ij}$ to the position which immediately precedes $B_1$. As $T_{ij}$ is pushed
            through each such $T_{pn}$, augment $F_{pn}$ according to Lemma 3.
```

The integrated repair algorithm using Algorithm 14 to rewrite a history and the compensation approach to prune the rewritten history is specified as follows.

Algorithm 15 Repair a history of sagas by can-follow rewriting
Input: the serial history H^s to be repaired
Output: a repaired history H_r^s which consists of only good saga transactions.
Method:
1. Rewrite H^s using Algorithm 14 [‡].

[†] The *lifetime* of a saga begins when the saga is initiated, and ends when the saga terminates (commits or aborts).

[‡] It is possible that in H_e^s one part of a saga S_i is in H_r^s and the other part of S_i is in $H_e^s - H_r^s$.

2. Do compensation from the end to the beginning of H_e^s until B_1 is compensated. When the final step T_{pn} of a saga S_p is to be compensated

- First, execute $T_{pn}^{(-1,F_{pn})}$ to compensate $T_{pn}^{F_{pn}}$. The codes and input parameters of $T_{pn}^{(-1,F_{pn})}$ are got from the identification of the compensating transaction of T_{pn} included in the *end-transaction call* of T_{pn}. Note that F_{pn} has already been computed after H^s was rewritten.

- Second, execute the sequence $T_{p(n-1)}^{(-1,\emptyset)}$, ..., $T_{pq}^{(-1,\emptyset)}$ to compensate all the other steps of S_p which are not in H_r^s. The codes and input parameters of these compensating transactions are got in the same way as of $T_{pn}^{(-1,F_{pn})}$.

The correctness of Algorithm 15 is specified as follows.

Theorem 16 Algorithm 15 is correct in the sense that H_r^s is consistent after step 1, and the repair results in the same state as generated by re-executing H_r^s.

Proof: It is clear that in Algorithm 14 a step T_{ij} will not be moved into H_r^s unless every step between T_{i1} and T_{ij} can be moved into H_r^s. For a step T_{ij}, if S_i is a good saga, and each step between T_{i1} and T_{ij} (including T_{ij}) can be moved into H_r^s, then we say T_{ij} is an *unaffected step*, otherwise, we say T_{ij} is an *affected step.*

We propose another approach to repair H^s which is clearly correct. It works as follows: Scan H^s backward from the end to the beginning:

- If a bad final step B_{in} is met, execute the sequence $B_{in}^{(-1,F_{in})}$, $B_{i(n-1)}^{(-1,\emptyset)}$, ..., $B_{i1}^{(-1,\emptyset)}$ on the final state of the current history. This can remove the effects of B_i from the current history because at this point all the steps to the right of B_{in} are unaffected steps. All the bad or affected steps to the right of B_{in} have already been compensated. So B_{in} can follow every step following it, thus B_{in} can be moved to the end of the current history without changing the final state of the current history if F_{in} is computed according to Lemma 3, therefore, according to the Compensation Property, after $B_{in}^{F_{in}}$ is compensated executing $B_{i(n-1)}^{(-1,\emptyset)}$, ..., $B_{i1}^{(-1,\emptyset)}$ can compensate the other steps.

- If an affected final step T_{in} is met, assume T_{ip} is the last unaffected step in S_i, execute the sequence $T_{in}^{(-1,F_{in})}$, $T_{i(n-1)}^{(-1,\emptyset)}$, ..., $T_{i(p+1)}^{(-1,\emptyset)}$. For similar reasons to the above case, this can remove the effects of all the affected steps of S_i.

It is clear that the above approach results in H_r^s. So H_r^s is consistent. Since the above approach executes the same set of fixed compensating transactions on the final state of H^s as Algorithm 15, and it executes these fixed compensating transactions in the same order, so Algorithm 15 results in the same state as generated by re-executing H_r^s. □

Algorithm 11 and Algorithm 12 can also be adapted to rewrite a history of sagas. The adapted algorithms are specified as follows. For brevity, and to highlight the differences between these algorithms, we describe only the modifications to Algorithm 11 and to Algorithm 12, respectively.

Algorithm 16 Rewrite a history of sagas by can-follow and can-precede rewriting
Method: Scan H^s forward from the first good step after B_{11} until the end of H^s, for each step T_{ij}

```
    case $S_i \in \mathbf{G}$
        if there is a step of $S_i$ which stays between $B_1$ and $T_{ij}$
            Skip $T_{ij}$;
        elseif the final step $T_{pn}$ of every saga $S_p$ which stays between $B_1$ and $T_{ij}$
         (including $B_1$) can follow $T_{ij}$, or $T_{ij}$ can precede $T_{pn}^{F_{pn}}$
            Move $T_{ij}$ to the position which immediately precedes $B_1$.
```

Algorithm 17 Rewrite a history of sagas by can-follow, can-precede, cover and invert rewriting
Method: Scan H^s forward from the first good step after B_{11} until the end of H^s, for each step T_{ij}

```
    case $S_i \in \mathbf{G}$
        if there is a step of $S_i$ which stays between $B_1$ and $T_{ij}$
            Skip $T_{ij}$;
        elseif the final step $T_{pn}$ of every saga $S_p$ which stays between $B_1$ and $T_{ij}$
         (including $B_1$) can follow $T_{ij}$, or $T_{ij}$ can precede $T_{pn}^{F_{pn}}$,
         or $T_{ij}$ covers $T_{pn}^{F_{pn}}$, or $T_{ij}$ inverts $T_{pn}^{F_{pn}}$
            Move $T_{ij}$ to the position which immediately precedes $B_1$.
```

The correctness of the repair based on Algorithm 16 or Algorithm 17 is specified in the following theorem. The proof is similar to that of Theorem 16, thus omitted.

Theorem 17 The repair based on Algorithm 16 is correct in the sense that Theorem 16 still holds even if the rewriting step (step 1) of Algorithm 15 is done by Algorithm 16. The repair based on Algorithm 17 is correct in the sense that Theorem 16 still holds after Algorithm 15 is modified as follows:

- The rewriting step (step 1) is done by Algorithm 17;

- In step 2, when an affected step $T_{i(n-1)}$ of a saga S_i is scanned [§], assume T_{ip} is the last unaffected step in S_i, execute the sequence $T_{i(n-1)}^{(-1,\emptyset)}$, ..., $T_{i(p+1)}^{(-1,\emptyset)}$.

6.6.3 Detecting Can-Follow, Can-Precede, Cover and Invert Relationships between Transactions

In Section 6.6.2, the repair based on Algorithm 14, Algorithm 16 and Algorithm 17 cannot be enforced without first capturing the *can-follow*, *can-precede*, *cover*, and *invert* relationships between saga transactions.

Given a history of sagas, the can-follow relationships between the saga transactions in the history depend on the readset-writeset relationships between these transactions. The write set of a transaction T_i can be got from the traditional log where every write operation is recorded. However, the read information of T_i we can get from the logs for traditional recovery purposes such as physical logs, physiological logs, and logical logs [GR93], is usually not enough to generate the read set. Therefore, the efficient maintenance of read information is a critical issue. In particular, there is a tradeoff between the extra cost we need to pay besides that of traditional recovery facilities and the guaranteed availability of read information. The read information can be captured in several ways, for example

- Augment the write log to incorporate read information. There are basically two ways: one is appending the read record $[T_i, x]$ to the log every time when T_i reads an item x. The other way is first keeping the set of items read by T_i in another place until the time when T_i is going to commit. At this point, the read set of T_i can be forced to the log as one record.

  Although keeping read information in the log will not cause more forced I/O, it does consume more storage. Another problem with the approach lies in the fact that almost all present database systems keep only update(write) information in the log. Thus adding read records to the log may cause the redesign of the current recovery mechanisms.

- Extract read sets from the profiles and input arguments of transactions. Compared with the read log approach, when transaction profiles (or codes) are available, each transaction just needs to store its input parameters, which are often much smaller in size than the read set. More important, instead of putting the input parameters in the log, each transaction can store the parameters in a specific user database, thus the repair module can be completely isolated from the traditional recovery module. In this way, our repair model can be implemented on

[§] This may happen because T_{in} may have already been covered or inverted.

top of the Saga model without modifying the internals of the DBMS on which the Saga model is implemented.

This approach captures read information without the need to modify DBMS internals. However, it usually can only achieve a complete repair, but not an exact repair. That is, the effects of all bad transactions will be removed, but the effects of some unaffected good transactions may sometimes be removed also since in many situations the approach can only get an approximate read set.

- Although traditional logging only keeps write information, more and more read information can be extracted from the log, particularly when more operation semantics are kept in the logs. Traditional physical(value) logging keeps the before and after images of physical database objects(i.e., pages), so we only know that some page is read. In addition, a page is normally too large a unit to achieve a fine repair. Physiological logging keeps only the update to a record(tuple) within one and only one page, so we know that this record should be in the read set, which is much finer than physical logging. Logical logging keeps more operation semantics than the other two logging approaches. Conceptually logical logs can keep track of all the read information of a transaction, though this is not supported by current database systems. However, logical logging attracts substantial industrial and research interests. In system R, SQL statements are put into the log as logical records; In [LT98], logical logs can be a function, like `x=sum(x,y)`, and `swap(x,y)` etc.. In both situations, we get more read information than other logging methods.

  In long duration transaction models([GMS87], [WR91]), or in multilevel transaction models ([WHBM90], [Lom92]), it is possible to extract the read information of transaction (subtransaction) T from its compensation log records, where the action of T's compensating transaction is recorded.

The *can-precede*, *cover*, and *invert* relationships between transactions are based on the semantics of transactions, and they can be captured in a similar way to commutativity[LMWF94, Wei88, Kor83, SKPO88], and recoverability[BK92]. In order to capture these relationships, the profile (or code) and input arguments of each transaction must be available. In the Saga model, several possible solutions to the problem of saving code reliably are proposed[GMS87], therefore, these relationships can be reliably captured in the Saga model.

For a *canned* system with limited number of transaction classes and fixed code for each transaction class, the *can-follow*, *can-precede*, *cover*, and *invert* relationships between saga transactions can be detected according to the corresponding relationships between transaction classes. Although detecting these relationships between two transaction classes usually needs more effort than detecting these relationships between two transactions, after this is done with all the transaction classes, detecting these relationships between transactions of these classes can be much easier in many situations.

For example, in a bank a deposit transaction (denoted $dep(a_i, m)$) which deposits m amount of money into account a_i can follow a withdraw transaction (denoted $wit(a_j, n)$) which withdraws n amount of money from account a_j only if they access different accounts, that is, $a_i \neq a_j$. Therefore, given the can-follow relationship between the deposit transaction class and the withdraw transaction class, the can-follow relationship between a deposit transaction and a withdraw transaction can be detected without the need to check the readset-writeset relationship between the two transactions, checking their input parameters is enough.

6.6.4 Fix Information Maintenance

It is clear that Lemma 4 can be used in Algorithm 14, Algorithm 16 and Algorithm 17, to capture fixes. For a transaction T_i, there are two methods to get $T_i.readset - T_i.writeset$: one is to first get the readset and writeset of T_i after an execution history is generated using the approaches proposed in Section 6.6.3, then compute $T_i.readset - T_i.writeset$; the other is what we have proposed in Section 6.2, that is, let each transaction T_i write the set $T_i.readset - T_i.writeset$ as a record to the database when it is executed, then when we rewrite H^s all the fixes can be directly got from the database.

It should be noticed that in the situations where the read and write sets of T_i have to be firstly captured in order to detect the can-follow relationships between T_i and some other transaction, the first method is more efficient; In contrast, when all the necessary can-follow relationships between T_i and other transactions can be detected without the need to check the readset-writeset relationships between T_i and these transactions, for example, when these relationships can be directly got from the can-follow relationships between the corresponding transaction classes, the second method is more efficient.

SECTION 7

Discussions and Conclusions

7.1 Discussion

7.1.1 Relevant Security Contexts

Our repair model can be applied to many kinds of secure database systems to enhance their survivability. However, the main factors on which the applicability of our model to a secure database system is dependent, such as (1) the characteristics of the database, i.e., whether it is single-version or multiversion, (2) the concurrency control protocol and the characteristics of the histories produced by it, and (3) the recovery protocol and the characteristics of the logs produced by it, are closely relevant to the security model and architecture of the system.

For a single-level secure database system where every subject (transaction) and object (data item) are within the same security class, traditional concurrency control protocols such as two-phase locking (2PL), and recovery protocols such as write-ahead logging (WAL), can be directly used without causing any security policy violations, no matter which kind of security model (i.e., access-matrix model[Lam74], role-based access control model[SCFY96], type-based access control model[San92], or flexible access-control model[JSS97]) is enforced. Since serializable histories are generated by most of the current single-level systems, so our repair model can be directly applied to single-level systems in most cases. However, there are some systems where each data item has multiple versions, and one-copy serializable histories are generated instead. Since an one-copy serializable history is view equivalent to a serial single-version history[BHG87], our model can be used to repair the one-copy serializable history by rewriting the equivalent serial history. However, it should be noticed that pruning a rewritten history in multiversion databases is usually more complicated because during pruning we need to decide for a (dirty) data item which version should be read, which version should be updated, and which version should be discarded (i.e., the versions created by bad transactions can just be discarded). Detailed pruning algorithms are out of the scope of the paper.

For a multilevel secure (MLS) database system, traditional concurrency control and recovery protocols, however, are usually not enough to satisfy security requirements[AJB97], especially, they can cause signaling channels from high level processes to low level processes. Therefore, secure transaction processing is required. Most of the recent research and development in secure concurrency control can be categorized into two different areas: one based on *kernelized* architecture and the other based on *replicated* architecture. These two are among the number of architectures proposed by the Woods Hole study group[oMDMSBC83] to build multilevel secure DBMSs with existing DBMS technology instead of building a trusted DBMS from scratch.

For kernelized architecture, several kinds of secure concurrency control protocols are proposed: (1) In [MJ93, JMR97], several secure lock-based protocols are proposed. Although they do not always produce serializable schedules, our repair model can be directly applied to every serializable history generated by them. Extending our model to repair those non-serializable schedules is out of the scope of the paper. (2) In [AJ92], two secure timestamp-based protocols are proposed. Although they produce only serializable histories to which our model can be directly applied, they are prone to starvation. In [JA92], a single-level timestamp-based scheduler is proposed which is secure and free of starvation. Although it produces one-copy serializable histories, our model can still be directly used to rewrite these histories (the reason is mentioned above). (3) In [AJB96, JA92, AJB97], three weaker notions of correctness, namely, *levelwise* serializability, *one-item read* serializability, and *pairwise* serializability, are proposed to be used as alternative for one-copy serializability such that the nature of integrity constraints in MLS databases can be exploited to improve the amount of concurrency. Extending our model to repair levelwise, one-item read, and/or pairwise serializable histories is out of the scope of the paper.

For replicated architecture, several secure concurrency control protocols are proposed in [JK90, MJS91, Cos92, CM92]. Since they all produce one-copy serializable histories, so our model can be directly applied to rewrite these histories.

In [KT90], a scheduler is proposed which is secure and produces one-copy serializable histories to which our model can be applied. However, it uses a multilevel scheduler which, therefore, has to be trusted, thus it is only suitable for the *trusted subject* architecture.

Since in our repair model serial orders among transactions are captured from the log, so the applicability of our model is affected by logging protocols. In [PKP97], a multilevel secure log manager is proposed to eliminate such covert channels as *insert* channels and *flush* channels which are caused by traditional logging protocols. Although *Logical Log Sequence Numbers* (LLSN) instead of physical *Log Sequence Numbers* (LSN) are provided in [PKP97] to eliminate insert channels, we can still extract serial orders from the log because records of transactions within different security classes are still kept in the same log, and LLSNs can be translated to physical LSNs internally by the log manager. Moreover, since the mechanisms proposed to eliminate flush channels will not change the structure of the log, so our model can be directly applied to a system with such a log manager.

7.1.2 Other Issues

One criticism of the applicability of the method may be that if a bad transaction B_i is detected too late, that is, if the *latency time* of B_i is too long, then there can be too many affected good transactions to deal with, especially when they have caused further effects to the real world. For example, some real world decisions could be based on these affected transactions. At this situation, 'manual' recovery actions may be necessary.

We counter this augment by noting that the latency time of B_i is usually related to the amount of transactions affected by B_i. The more transactions affected by B_i, the more proofs of B_i's malicious actions can be collected by the intrusion detector, hence the shorter the latency time of B_i. Therefore, even if the *latency time* of B_i is very long, the amount of transactions affected by B_i may not be too large in many circumstances. At this situation, the algorithm may need more time since it needs to scan a long history, but the pruning may still be a short process if most of the transactions in the history are unaffected. Although the compensation approach may not be practical when the history is very long and the codes for compensating transactions have to be kept in the log, it can be used in almost all canned systems, which are very general in real world where the codes for transactions and compensating transactions are fixed for each transaction class. As the techniques of intrusion detection are advanced, the latency time of a bad transaction should become shorter, so our repair model will apply to more situations.

As to the criticism that manual recovery actions can be necessary, note that when damage has been caused, the effects of these affected transactions to the real world are already there. No matter whether the history is repaired or not, some action to compensate these undesirable effects is required. In the real world, such manual recovery actions are basically unavoidable. Therefore, repairing the database such that a consistent database state where no effects of bad transactions are there could be generated can be viewed as a separate issue from manual recovery. In addition, our rewriting methods can help users to assess the degree of damages because $\mathbf{B} \cup \mathbf{AG}$ can be identified. Therefore, the security administrator can know on which transactions (or on which customers) such manual recovery actions should be enforced.

7.2 Contributions

There are three areas where this research made contributions. First, this report proposes two novel recovery models to bridge the theoretical gap between classical database recovery theory where only uncommitted transactions can be undone, and trusted recovery practice where operations with the same (operational) semantics as traditional undos are needed to remove the effects of such committed transactions as malicious transactions and affected benign transactions (For simplicity, we use the same word, namely 'undo', to denote such operations). In particular, this report proposes (1) a *flat-transaction* recovery model where committed transactions are 'undone' by building and executing a specific type of transactions, namely, *undo* transactions, and (2) a *nested-transaction* model where a flat commercial history is virtually extended to a two-layer nested structure where originally committed transactions turn out to be subtransactions hence traditional undo operations can be directly applied to the model without violating the durability property.

Second, this report provides a family of syntactic recovery algorithms that, given a specification of malicious, committed transactions, unwinds the effects of each malicious transaction, along with the effects of any benign transaction that depends, directly or indirectly on a malicious transaction. Significantly, the work of the remaining benign transactions is saved. The first algorithm yields coldstart semantics; the database is unavailable during repair. The second algorithm yields warmstart semantics; normal use may continue during repair, although some degradation of service may be experienced by some transactions. Moreover, this report outlines various possibilities for maintaining read-from dependency information. Although direct logging of transaction reads has the virtue of simplicity, the performance degradation of such an approach may be too severe in some cases. For this reason, this report shows that offline analysis can efficiently meet the need for establishing read-from dependency information. This report illustrates the practicality of such an approach via a study on standard benchmarks.

Third, this report presents an algorithm that rewrites an execution history for the purpose of backing out malicious transactions. Good transactions that are affected, directly or indirectly, by malicious transactions complicate the process of backing out undesirable transactions. This report shows that the prefix of a rewritten history produced by the algorithm serializes exactly the set of unaffected good transactions, thus is equivalent to using a write-read dependency graph approach. The suffix of the rewritten history includes special state information to describe affected good transactions as well as malicious transactions. This report describes techniques that can extract additional good transactions from the latter part of a rewritten history. The latter processing saves more good transactions than is possible with a dependency-graph based approach or a commutativity based approach to recovery.

It is also shown that besides recovery from malicious transactions, our recovery approaches can also be extended to may other applications such as malicious user isolation, system upgrades, optimistic replication protocols, and replicated mobile databases.

7.3 Future Research

Based on the research work in this report, we propose the following future research directions.

7.3.1 Trusted Recovery with Bounded Inconsistency

It is clear that every rewriting algorithm proposed in Chapter 6 has the following two properties: (1) it always works on a consistent history *; (2) every rewriting operation performed by the algorithm always transforms a consistent history to another consistent history. However, we found that by tolerating some degree of inconsistency in rewriting histories the work of more good transactions can be saved. The cost is that after a consistent history is repaired, it may not be consistent any more.

To illustrate the idea, consider a banking system where a customer can deposit (withdraw) money into (from) his/her accounts, but with the integrity constraint that the balances of his/her accounts can not be negative. It is clear that a deposit transaction (denoted $dep(a_i, m)$) which deposits m amount of money into account a_i can precede any other deposit transactions. However, according to Definition 8, a withdraw transaction (denoted $wit(a_j, n)$) can not precede $dep(a_i, m)$ if $a_i = a_j$ and the balance of a_i in the before state of $dep(a_i, m)$ (denoted s_b) is less than n, because at this point $wit(a_j, n)dep(a_i, m)$ is not defined on s_b since the execution of $wit(a_j, n)$ on s_b makes the database state inconsistent. Hence, when we rewrite such a history with $dep(a_i, m)$ followed by $wit(a_j, n)$ and with s_b as the before state of $dep(a_i, m)$, if $dep(a_i, m)$ is a bad transaction, then the work of $wit(a_j, n)$ can not be saved.

However, if we can tolerate a bounded degree of inconsistency, for example, allowing a balance greater than -5000, then in the above situation $wit(a_j, n)$ can precede $dep(a_i, m)$ if the difference between the value of a_j in s_b and n is less than 5000. Therefore, the work of $wit(a_j, n)$ can be saved in the above example.

In order to enable trusted recovery with bounded inconsistency, several critical issues have to be addressed:

- To enable a transaction to be executed on an inconsistent database state, or to enable the transaction to transform a consistent state to an inconsistent one, the preconditions, or even the action, of the transaction may need to be modified. How to formalize the modification is a critical issue.

*We say a history H is *consistent*, if the before and after states of each transaction in H are both consistent, no matter whether H has a transaction associated with a non-empty fix or not; otherwise, we say H is *inconsistent*.

- After a transaction is modified, the can-follow, can-precede, commute backward through, and commute relationships between the transaction and other transactions may have to be reidentified. Formalizing and automatizing the process of reidentification is a critical issue.

- In order to enable an inconsistent rewriting operation which exchanges the order of two transactions, T_i and T_j, during rewriting a consistent history, the modified version(s) of T_i, or T_j, or both, may have to be introduced in the rewritten history. Thus how to formalize and reason the relationship between the history before the rewriting operation is performed and the history after the rewriting operation is a critical issue that we have to address. The correctness of rewriting with bounded inconsistency depends on it.

7.3.2 Extension to Multilevel Secure Systems

As mentioned in Section 7.1.1, the applicability of our repair model to a secure database system is closely relevant to the security model and architecture of the system. Although our model can be directly applied to most single-level secure systems, there are many multilevel secure database systems where our repair model has to be extended.

- In static repair, since the repair manager can be the only user process running during the process of trusted recovery, so there is no information disclosure during the repair. However, in dynamic repair, the fact that the repair manager is usually running together with many other user processes implies that in a system where the kernelized architecture is used, there can be signaling channels from high-level processes to low-level ones. How to build a single-level repair manager without introducing signaling channels has to be addressed.

- Although our model can be directly used to rewrite one-copy serializable histories generated by secure concurrency control protocols which exploit multiple versions of a data item, pruning a rewritten history in multiversion databases is usually more complicated because during pruning we need to decide for a (dirty) data item which version should be read, which version should be updated, and which version should be discarded. This issue has to be addressed.

- In [AJB96, JA92, AJB97], three weaker notions of correctness, namely, *levelwise* serializability, *one-item read* serializability, and *pairwise* serializability, are proposed to be used as alternative for one-copy serializability such that the nature of integrity constraints in MLS databases can be exploited to improve the amount of concurrency. Extending our model to repair levelwise, one-item read, and/or pairwise serializable histories is another critical issue.

7.3.3 Extension to Distributed Database Systems

A *distributed database* (DDB) consists of several logical objects that are physically located at different sites (or nodes). Each site consists of an independent processor connected via communication links to other sites. Transaction executing in these systems may require to access (either update or retrieve) data objects from more than one site. The site at which a transaction originates is usually referred to as the *coordinator* and other sites participating in the execution are called *subordinate* sites.

In a distributed database system, data are partitioned and stored across several nodes which are connected by a network. Therefore, the dependency-graph of the global history generated from the system can not be mined from a local log in most situations. Instead, we may have to combine the information recorded in every local log to compute the global dependency-graph based on which syntactic repair can be achieved. Moreover, the can-follow relationship between two distributed transactions depends also on the read and write behavior of these transactions at multiple sites. Therefore, integrating multiple local logs is an issue that has to be addressed.

As mentioned in Section 7.3.2, how to build a single-level dynamic repair manager without introducing signaling channels in a multilevel secure database system is a critical issue. Similarly, how to build a single-level dynamic repair manager without introducing signaling channels in a distributed multilevel secure database system is also a critical issue. The difference is that in distributed MLS systems integration of secure concurrency control protocols, i.e., S2PL [JM93], with atomic commit protocols, i.e., *early prepare* (EP), may not guarantee serializability [JMB94], thus corresponding secure commit protocols have to be developed.

BIBLIOGRAPHY

[AJ92] P. Ammann and S. Jajodia. A timestamp ordering algorithm for secure, single-version, multi-level databases. In C. Landwehr and S. Jajodia, editors, *Database Security, V: Status and Prospects*, pages 23–25. Amsterdam: North Holland, 1992.

[AJB96] V. Atluri, S. Jajodia, and E. Bertino. Alternative Correctness Criteria for Concurrent Execution of Transactions in Multilevel Secure Databases. *IEEE Transactions on Knowledge and Data Engineering*, 8(5):839–854, October 1996.

[AJB97] V. Atluri, S. Jajodia, and E. Bertino. Transaction Processing in Multilevel Secure Databases with Kernelized Architecture: Challenges and Solutions. *IEEE Transactions on Knowledge and Data Engineering*, 9(5):697–708, 1997.

[AJM95] P. Ammann, S. Jajodia, and P. Mavuluri. On the fly reading of entire databases. *IEEE Transactions on Knowledge and Data Engineering*, 7(5):834–838, October 1995.

[AJMB97] P. Ammann, S. Jajodia, C.D. McCollum, and B.T. Blaustein. Surviving information warfare attacks on databases. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 164–174, Oakland, CA, May 1997.

[AJR97] P. Ammann, S. Jajodia, and I. Ray. Applying formal methods to semantic-based decomposition of transactions. *ACM Transactions on Database Systems*, 1997. To appear.

[ALS78] T. Anderson, P. A. Lee, and S. K. Shrivastava. A Model of Recoverability in Multilevel Systems. *IEEE Transactions on Software Engineering*, 4(6):486–494, November 1978.

[BBG83] A. Borg, J. Baumbach, and S. Glazer. A message system supporting fault tolerance. In *Proceedings of the Ninth ACM Symposium on Operating System Principles*, pages 90–99, oct 1983.

[Ber88] P. A. Bernstein. Sequoia: A Fault-Tolerant Tightly Coupled Multiprocessor for Transaction Processing. *IEEE Computer*, 21(2):37–45, February 1988.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[BK92] B.R. Badrinath and Ramamritham Krithi. Semantics-based concurrency control: Beyond commutativity. *ACM Transactions on Database Systems*, 17(1):163–199, March 1992.

[BL76] D.E. Bell and L.J. LaPadula. Secure computer systems: Unified exposition and multics interpretation. Technical Report MTR-2997, The Mitre Corporation, Bedford, MA, March 1976.

[cer95] Cert coordination center 1995 annual report. Technical report, the CERT Coordination Center, 1995. http://www.cert.org/pub/annual-reports/cert_rpt_95.html.

[cer96] Cert coordination center 1996 annual report. Technical report, the CERT Coordination Center, 1996. available at http://www.cert.org.

[CM92] O. Costich and J. McDermott. A multilevel transaction problem for multilevel secure database systems and its solution for the replicated architecture. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 192–203, Oakland, CA, 1992.

[Cos92] O. Costich. Transaction processing using an untrusted scheduler in a multilevel secure database with replicated architecture. In C. Landwehr and S. Jajodia, editors, *Database Security, V: Status and Prospects*, pages 173–189. Amsterdam: North Holland, 1992.

[Dat83] C. J. Date. *An Introduction to Database Systems, Volume II.* Addison–Wesley, Reading, MA, 1983.

[Dat95] C. J. Date. *An Introduction to Database Systems, Sixth Edition.* Addison–Wesley, Reading, MA, 1995.

[Dav84] S. B. Davidson. Optimism and consistency in partitioned distributed database systems. *ACM Transactions on Database Systems*, 9(3):456–581, September 1984.

[Den83] Dorothy E. Denning. *Cryptography and Data Security.* Addison-Wesley, Reading, MA, 1983.

[Den87] D. E. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, 13(2):222–232, February 1987.

[dod96] Information security: Computer attacks at department of defense pose increasing risks. Technical Report AIMD-96-84, General Accounting Office, DoD, 1996. available at http://www.nsi.org/Library/Compsec/infosec.txt.

[fbi97] Computer crime and security survey. Technical report, Computer Security Institute, 1997. available at http://www.gocsi.com/scu.preleas2/htm.

[GHOS96] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 173–182, Montreal, Canada, 1996.

[GL91] T.D. Garvey and T.F. Lunt. Model-based intrusion detection. In *Proceedings of the 14th National Computer Security Conference*, Baltimore, MD, October 1991.

[GM83] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.

[GMS87] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 249–259, San Francisco, CA, 1987.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, Inc., 1993.

[Gra93] J. Gray, editor. *The Benchmark Handbook for Database and Transaction Processing Systems.* Morgan Kaufmann Publishers, Inc., 2 edition, 1993.

[GS96] S. Garfinkel and E. H. Spafford. *Practical UNIX Security.* O'Reilly & Associates, Inc., CA, 1996.

[GSM96] R. Graubart, L. Schlipper, and C. McCollum. Defending database management systems against information warfare attacks. Technical report, The MITRE Corporation, 1996.

[HR83] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *Computing Surveys*, 15(4):287–318, 1983.

[HR98] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. In V. Kumar and M. Hsu, editors, *Recovery Mechanisms in Database Systems*, pages 16–55. Prentice Hall PTR, 1998.

[IKP95] K. Ilgun, R.A. Kemmerer, and P.A. Porras. State transition analysis: A rule-based intrusion detection approach. *IEEE Transactions on Software Engineering*, 21(3):181–199, 1995.

[Ilg93] K. Ilgun. Ustat: A real-time intrusion detection system for unix. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1993.

[JA92] S. Jajodia and V. Atluri. Alternative correctness criteria for concurrent execution of transactions in multilevel secure databases. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 216–224, Oakland, CA, 1992.

[JK90] S. Jajodia and B. Kogan. Transaction processing in multilevel secure databases using replicated architecture. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 360–368, Oakland, CA, 1990.

[JLM98] S. Jajodia, P. Liu, and C.D. McCollum. Application-level isolation to cope with malicious database users. In *Proceedings of the 14th Annual Computer Security Application Conference*, pages 73–82, Phoenix, AZ, December 1998.

[JM93] S. Jajodia and C. McCollum. Using two-phase commit for crash recovery in federated multilevel secure database management systems. In C. E. Landwehr et al., editor, *Dependable Computing and Fault Tolerant Systems*, pages 365–381. Springer-Verlag, 1993.

[JMB94] S. Jajodia, C. D. McCollum, and B. T. Blaustein. Integrating concurrency control and commit algorithms in distributed multilevel secure databases. In T. F. Keefe and C. E. Landwehr, editors, *Database Security, VII: Status and Prospects*, pages 109–121. Amsterdam: North Holland, 1994.

[JMR97] S. Jajodia, L. Mancini, and I. Ray. Secure locking protocols for multilevel database management systems. In P. Samarati and R. Sandhu, editors, *Database Security X: Status and Prospects.* London: Chapman & Hall, 1997.

[JSS97] S. Jajodia, P. Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 31–42, Oakland, CA, 1997.

[JV94] H. S. Javitz and A. Valdes. The nides statistical component description and justification. Technical Report A010, SRI International, March 1994.

[JZ90] D. B. Johnson and W. Zwaenepoel. Recovery in distributed systems. *Journal of Algorithms*, 11(3):462–491, September 1990.

[KLS90] H.F. Korth, E. Levy, and A. Silberschatz. A formal approach to recovery by compensating transactions. In *Proceedings of the International Conference on Very Large Databases*, pages 95–106, Brisbane, Australia, 1990.

[Kor83] Henry F. Korth. Locking primitives in a database system. *Journal of the ACM*, 30(1):55–79, January 1983.

[KQ72] P. J. Kennedy and T. M. Quinn. Recovery strategies in the no. 2 electronic switching system. In *Digest of Papers: 1972 International Symposium on Fault-Tolerant Computing*, pages 165–169, Newton, MA, 1972.

[KT90] T. F. Keefe and W. T. Tsai. Multiversion concurrency control for multilevel secure database systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 369–383, Oakland, CA, 1990.

[LA90] P.A. Lee and T. Anderson. *Fault Tolerance: Principles and Practice, Second edition.* Springer-Verlag, Wien, Austria, 1990.

[LA94] H. V. Leong and D. Agrawal. Using message semantics to reduce rollback in optimistic message logging recovery schemes. In *Proceedings of the 14th International Conference on Distributed Computing Systems*, pages 227–234, 1994.

[Lam74] B. W. Lampson. Protection. *ACM Operating Systems Review*, 8(1):18–24, January 1974.

[LMWF94] N. Lynch, M. Merritt, W. Weihl, and A. Fekete. *Atomic Transactions.* Morgan Kaufmann, 1994.

[Lom92] D.B. Lomet. MLR: A recovery method for multi-level systems. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 185–194, San Diego, CA, June 1992.

[LT98] D. Lomet and M. R. Tuttle. Redo recovery after system crashes. In V. Kumar and M. Hsu, editors, *Recovery Mechanisms in Database Systems*, chapter 6. Prentice Hall PTR, 1998.

[Lun93] T.F. Lunt. A Survey of Intrusion Detection Techniques. *Computers & Security*, 12(4):405–418, June 1993.

[LWJ98] J. Lin, X. S. Wang, and S. Jajodia. Abstraction-based misuse detection: High-level specifications and adaptable strategies. In *Proceedings of the 11th IEEE Computer Security Foundations Workshop*, Rockport, Massachusetts, June 1998.

[MG96a] J. McDermott and D. Goldschlag. Storage jamming. In D.L. Spooner, S.A. Demurjian, and J.E. Dobson, editors, *Database Security IX: Status and Prospects*, pages 365–381. Chapman & Hall, London, 1996.

[MG96b] J. McDermott and D. Goldschlag. Towards a model of storage jamming. In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 176–185, Kenmare, Ireland, June 1996.

[MHL$^+$92] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: A transaction recovery method supporting fine-granularity locking. *ACM Transactions on Database Systems*, 17(1):94–162, March 1992.

[MHL94] B. Mukherjee, L. T. Heberlein, and K.N. Levitt. Network intrusion detection. *IEEE Network*, pages 26–41, June 1994.

[MJ93] J. McDermott and S. Jajodia. Orange locking: Channel-free database concurrency control. In B. M. Thuraisingham and C. E. Landwehr, editors, *Database Security, VI: Status and Prospects*, pages 267–284. Amsterdam: North Holland, 1993.

[MJS91] J. McDermott, S. Jajodia, and R. Sandhu. A single-level scheduler for replicated architecture for multilevel secure databases. In *Proceedings of the 7th Annual Computer Security Applications Conference*, pages 2–11, San Antonio, TX, 1991.

[Mos85] J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* Information Systems Series. The MIT Press, Cambridge, Massachussetts, 1985.

[MPL92] C. Mohan, H. Pirahesh, and R. Lorie. Efficient and flexible methods for transient versioning of records to avoid locking by read-only transactions. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 124–133, San Diego, CA, June 1992.

[oMDMSBC83] Committee on Multilevel Data Management Security, Air Force Studies Board, and National Research Council. *Multilevel Data Management Security.* National Academy Press, Washington, D.C., March 1983.

[PK92] P.A. Porras and R.A. Kemmerer. Penetration state transition analysis: A rule-based intrusion detection approach. In *Proceedings of the 8th Annual Computer Security Applications Conference*, San Antonio, Texas, December 1992.

[PKH88] C. Pu, G. Kaiser, and N. Hutchinson. Split transactions for open-ended activities. In *Proceedings of the International Conference on Very Large Databases*, pages 26–37, Auguest 1988.

[PKP97] V. R. Pesati, T. F. Keefe, and S. Pal. The design and implementation of a multilevel secure log manager. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 55–64, Oakland, CA, 1997.

[Pu86] C. Pu. On-the-fly, incremental, consistent reading of entire databases. *Algorithmica*, 1(3):271–287, October 1986.

[PW72] W. W. Peterson and E. J. Weldon. *Error-Correcting Codes.* MIT Press, MA, 1972.

[Ran77] B. Randell. System structure for software fault tolerance. In R. T. Yeh, editor, *Current Trends in Programming Methodology*, pages 195–219. Prentice-Hall, 1977.

[RC97] Krithi Ramamritham and Panos K. Chrysanthis. *Advances in Concurrency Control and Transaction Processing.* IEEE Computer Society Press, Los Alamitos, CA, 1997.

[RLKL95] B. Randell, J.-C. Laprie, H. Kopetz, and B. Littlewood, eds. *Predictably dependable computing systems.* Springer-Verlag, Berlin, 1995.

[San92] R. S. Sandhu. The typed access matrix model. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 122–136, Los Alamitos, CA, 1992.

[SCFY96] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-Based Access Control Models. *IEEE Computer*, (2):38–47, February 1996.

[SG91] S.-P. Shieh and V.D. Gligor. A pattern oriented intrusion detection model and its applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1991.

[SG97] S.-P. Shieh and V.D. Gligor. On a pattern-oriented model for intrusion detection. *IEEE Transactions on Knowledge and Data Engineering*, 9(4):661–667, 1997.

[SKPO88] M. Stonebraker, R. Katz, D. Patterson, and J. Ousterhout. The design of XPRS. In *Proceedings of the International Conference on Very Large Databases*, pages 318–330, Los Angeles, CA, 1988.

[SY85] R. Strom and S. Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, 3(3):205–226, August 1985.

[TB82] D. J. Tayor and J. P. Black. Principles of Data Structure Error Correction. *IEEE Transactions on Computers*, 31(7):602–608, July 1982.

[Wei88] W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, December 1988.

[WHBM90] G. Weikum, C. Hasse, P. Broessler, and P. Muth. Multi-level recovery. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium of Principles of Database Systems*, pages 109–123, Nashville, Tenn, April 1990.

[WR91] H. Wachter and A. Reuter. The contract model. In A. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 219–263. Morgan Kaufmann Publishers, 1991.

[WS92] G. Weikum and H.-J. Schek. Concepts and applications of multilevel transactions and open nested transactions. In Ahmed K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, chapter 13. Morgan Kaufmann Publishers, Inc., 1992.

☆U.S. GOVERNMENT PRINTING OFFICE: 2001-610-055-10145

MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)

The advancement and application of Information Systems Science and Technology to meet Air Force unique requirements for Information Dominance and its transition to aerospace systems to meet Air Force needs.